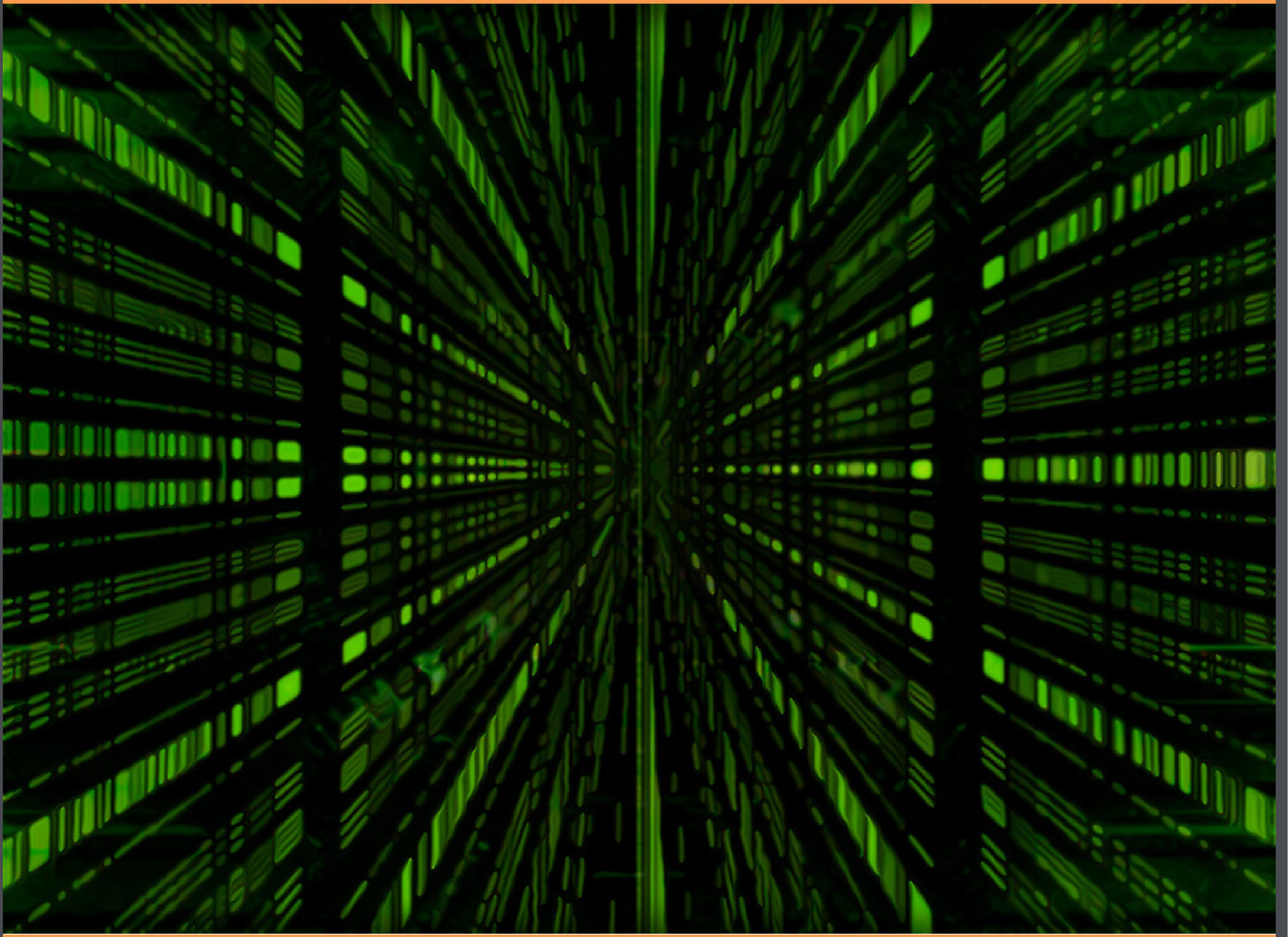
An Introduction to Relational Database Theory

Hugh Darwen



Hugh Darwen

An Introduction to Relational Database Theory

An Introduction to Relational Database Theory
4th edition
© 2014 Hugh Darwen & bookboon.com
ISBN 978-87-403-0777-1

This book is dedicated to the researchers at IBM United Kingdom's Scientific Centre, Peterlee, UK, in the 1970s, who designed and implemented the relational database language, ISBL, that has been my guide ever since.

Contents

HELT GRATIS!
for Skikk & Bank
DU FÅR BOKA HOS DNB

Preface

This book introduces you to the theory of relational databases, focusing on the application of that theory to the design of computer languages that properly embrace it. The book is intended for those studying relational databases as part of a degree course in Information Technology (IT). Relational database theory, originally proposed by Edgar F. Codd in 1969, is a topic in Computer Science. Codd's seminal paper (1970) was entitled *A Relational Model of Data for Large Shared Data Banks* (reference [5] in Appendix A).

An introductory course on relational databases offered by a university's Computer Science (or similarly named) department is typically broadly divided into a theory component and what we might call an "industrial" component. The "industrial" component typically teaches the language, SQL (Structured Query Language), that is widely used in the industry for database purposes, and it might also teach other topics of current significance in the industry. Although this book is *only about the theory*, I hope it will be interesting and helpful to you even if your course's main thrust is industrial.

In the companion book *SQL: A Comparative Survey* I show how the concepts covered in this book are treated in SQL, along with historical notes explaining how and when the treatments in question arose in the official version of that language. (*Aside*: SQL doesn't officially stand for anything, though it is usually assumed to stand for Structured Query Language. And the standard pronunciation is "ess-cue-ell", not "sequel", so a DBMS that supports it is an SQL DBMS, not a SQL DBMS.)

The book is directly based on a course of nine lectures that was delivered annually from 2004 to 2011 to undergraduates at the University of Warwick, England, as part of a 14-lecture module entitled *Fundamentals of Relational Databases*. The remaining five lectures of that module were on SQL. We encouraged the students to compare and contrast SQL with what they had learned in the theory part. We explained that study of the theory, and an example of a computer language based on that theory, should:

- enable them to understand the technology that is based on it, and how to use that technology (even if it is only loosely based on the theory, as is the case with SQL systems);
- provide a basis for evaluating and criticizing the current state of the art;
- illustrate of some of the generally accepted principles of good computer language design;
- equip those who might be motivated in their future careers to bring about change for the better in the database industry.

Examples and exercises in this book all use a language, **Tutorial D**, invented by the author and C.J. Date for the express purpose of teaching the subject matter at hand. Implementations of **Tutorial D**, which is described in reference [12], are available as free software on the Web. The one we use at the University of Warwick is called *Rel*, made by Dave Voorhis of the University of Derby. *Rel* is freely available at http://dbappbuilder.sourceforge.net/Rel.html.

This book is accompanied by *Exercises in Relational Database Theory,* in which the exercises given at the end of each chapter (except the last) are copied and a few further exercises have been added. Sample solutions to all the exercises are provided and the reader is strongly recommended to study these solutions (preferably *after* attempting the exercises!).

The book consists of eight chapters and two appendixes, as follows.

Chapter 1, **Introduction**, is based on my first lecture and gives a broad overview of what a database is, what a relational database is, what a database management system (DBMS) is, what a DBMS is expected to do, and how a relational DBMS does those things.

In Chapter 2, **Values, Types, Variables, Operators**, based on my second lecture, we look at the four fundamental concepts on which most computer languages are based. We acquire some useful terminology to help us talk about these concepts in a precise way, and we begin to see how the concepts apply to relational database languages in particular.

Relational database theory is based very closely on logic. Fortunately, perhaps, in-depth knowledge and understanding of logic are not needed. Chapter 3, **Predicates and Propositions**, based on my third lecture, teaches just enough of that subject for our present purposes, without using too much formal notation.

Chapter 4, **Relational Algebra—The Foundation**, based on material from lectures 4 and 5, describes the set of operators that is commonly accepted as forming a suitable basis for writing a special kind of expression that is used for various purposes in connection with a relational database—notably, queries and constraints.

Chapter 5, **Building on The Foundation**, describes additional operators that are defined in **Tutorial D** (lectures 5–6) to illustrate some of the additional kinds of things that are needed in a relational database language for practical purposes.

Chapter 6, **Constraints and Updating**, based on lecture 7, describes the operators that are typically used for updating relational databases, and the methods by which database integrity rules are expressed to a relational DBMS, declaratively, as constraints.

The final two chapters address various issues in relational database design. Chapter 7, **Database Design I: Projection-Join Normalization**, based on lectures 8 and 9, deals with one particularly important issue that has been the subject of much research over the years. Chapter 8, **Database Design II: Other Issues,** discusses some other common issues that are not so well researched. These are not dealt with in my lectures but they sometimes arise in the annual course work assigned to our students.

Note to Teachers

Over the years since 1970 there have been many books covering relational database theory. I have aimed for several distinguishing features in this one, namely:

1. Focusing, in the first six chapters, on the application of the theory in a computer language. (Choosing a language, for that purpose, that I co-designed myself might seem a little self-serving on my part. I would plead guilty to any such charge, but really there was no choice.)
2. Emphasizing the difference between relations *per se* and relation variables ("relvars"). Failure to do this in the past has resulted in all sorts of confusion.
3. Emphasizing the connection between the operators of the relational algebra and those of the first order predicate calculus.
4. Spurning Codd's distinction (and SQL's) between primary keys and alternate keys. As Codd himself originally pointed out, the choice of primary key is arbitrary.
5. In Chapter 7, on projection-join normalization, omitting details of normal forms that were defined in the early days but no longer seem useful, leaving just 6NF, 5NF, and BCNF. 2NF and 3NF are subsumed by the simpler BCNF, 4NF by the simpler 5NF. 1NF, not being a projection-join normal form, is dealt with (sort of) in Chapter 8. Domain-key normal form (DKNF) serves little purpose in practice and is not mentioned at all.
6. Also in Chapter 7, to study the normal forms in reverse order to that in which they are normally presented. I put 6NF first because it is the simplest and also the most extreme. More important to me was to deal with 5NF and join dependencies before BCNF and functional dependencies (though I do leave to the end discussion of those pathological cases where BCNF is satisfied but not 5NF).
7. In Chapters 7 and 8, taking care to include the integrity constraints that are needed in connection with each of the design choices under discussion; and, in Chapter 7, using those constraints to draw a clear distinction between decomposition as a genuine design choice and decomposition to correct design errors.

Topics that might reasonably be expected but are not covered include:

- relational calculus (after all, it is only a matter of notation)
- the so-called problem of "missing information" and approaches to that problem that involve major departures from the theory
- views (apart from a brief mention) and view updating (too controversial)
- DBMS implementation issues, performance and optimization, concurrency
- database topics that are not particular to relational databases—for example, security and authorization

Acknowledgments

Chris Date reviewed preliminary drafts of all the chapters and made many useful suggestions that I acted upon.

Erwin Smout carefully reviewed the first publication and reported many minor errors which have now been corrected. Further errors were subsequently reported by Gene Wirchenko, Wilhelm Steinbuss, Laith Alissa, Joe Abbate, and Bernard Lambeau. These have been corrected too. I am most grateful to all these people.

Ron Fagin saved me from making some egregious errors in connection with the definition of 5NF in Chapter 7. All remaining errors in this chapter and elsewhere in the book are, of course, my own.

When I started to prepare the material for my lectures at Warwick no implementation of **Tutorial D** existed and I was fully expecting that students would be doing my exercises on paper. Then an amazingly timely e-mail came out of the blue from Dave Voorhis, telling me about *Rel*. Even more fortuitously, Dave himself was (and is) working at another UK university no more than 60 miles away from mine, so we were able to meet face-to-face for the demo that confirmed *Rel's* usability for the purposes I had in mind.

My relationship with the Computer Science department at Warwick started many years ago when I was working for IBM. I am most grateful to Meurig Beynon, who first invited me to be a guest lecturer and has given me much support and encouragement ever since. Alexandra Cristea was my valued colleague on the database modules from 2006 to 2013 and I am grateful for her help and support too.

Fourth Edition Revisions

1. The **Tutorial D** examples and definitions have been revised where necessary to conform with Version 2 of that language. The revisions affect the operators EXTEND, SUMMARIZE, RENAME, UPDATE, GROUP, UNGROUP, WRAP, and UNWRAP, and also the WITH construct. The companion book *Exercises on Relational Database Theory* has been similarly revised and a few small changes were also needed in the other companion book, *SQL: A Comparative Survey*.
2. Appendix A of the first edition, dealing with differences between Version 1 and Version 2 of **Tutorial D**, clearly became surplus to requirements and has been dropped. In any case, both grammars are available at www.thethirdmanifesto.com.
3. Appendix B of previous editions becomes Appendix A, to which reference [8] has been added.
4. Chapter 7 has been revised to deal with a significant recent advance in the theory of normal forms given in reference [8].
5. The end notes of previous editions have been eliminated. In most cases their text has been incorporated into the main body.

1 Introduction

1.1 Introduction

This chapter gives a very broad overview of

- what a database is
- what a relational database is, in particular
- what a database management system (DBMS) is
- what a DBMS does
- how a relational DBMS does what a DBMS does

We start to familiarise ourselves with terminology and notation used in the remainder of the book, and we get a brief introduction to each topic that is covered in more detail in later sections.

1.2 What Is a Database?

You will find many definitions of this term if you look around the literature and the Web. At one time (in 2008), Wikipedia [1] offered this: "A structured collection of records or data." I prefer to elaborate a little:

A **database** is an *organized*, machine-readable collection of *symbols*, to be *interpreted* as a *true* account of some *enterprise*. A database is machine-updatable too, and so must also be a collection of *variables*. A database is typically available to a community of users, with possibly varying requirements.

The organized, machine-readable collection of symbols is what you "see" if you "look at" a database at a particular point in time. It is to be interpreted as a true account of the enterprise at that point in time. Of course it might happen to be incorrect, incomplete or inaccurate, so perhaps it is better to say that the account is *believed* to be true.

The alternative view of a database as a collection of variables reflects the fact that the account of the enterprise has to change from time to time, depending on the frequency of change in the details we choose to include in that account.

The suitability of a particular kind of database (such as relational, or object-oriented) might depend to some extent on the requirements of its user(s). When E.F. Codd developed his theory of relational databases (first published in 1969), he sought an approach that would satisfy the widest possible ranges of users and uses. Thus, when designing a relational database we do so without trying to anticipate specific uses to which it might be put, without building in biases that would favour particular applications. That is perhaps *the* distinguishing feature of the relational approach, and you should bear it in mind as we explore some of its ramifications.

1.3 "Organized Collection of Symbols"

For example, the table in Figure 1.1 shows an organized collection of symbols.

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3

Figure 1.1: An Organized Collection of Symbols

Can you guess what this tabular arrangement of symbols might be trying to tell us? What might it mean, for symbols to appear in the same row? In the same column? In what way might the meaning of the symbols in the very first row (shown in blue) differ from the meaning of those below them?

Do you intuitively guess that the symbols below the first row in the first column are all student identifiers, those in the second column names of students, and those in the third course identifiers? Do you guess that student S1's name is Anne? And that Anne is enrolled on courses C1 and C2? And that Cindy is enrolled on neither of those two courses? If so, what features of the organization of the symbols led you to those guesses?

Remember those features. In an informal way they form the foundation of relational theory. Each of them has a formal counterpart in relational theory, and those formal counterparts are the only constituents of the organized structure that is a relational database.

1.4 "To Be Interpreted as a True Account"

For example (from Figure 1.1):

StudentId	Name	CourseId
S1	Anne	C1

Perhaps those green symbols, organized as they are with respect to the blue ones, are to be understood to mean:

"Student S1, named Anne, is enrolled on course C1."

An important thing to note here is that only certain symbols from the sentence in quotes appear in the table—S1, Anne, and C1. None of the other words appear in the table. The symbols in the top row of the table (presumably column headings, though we haven't actually been told that) might help us to guess "student", "named", and "course", but nothing in the table hints at "enrolled". And even if those assumed column headings had been A, B and C, or X, Y and Z, the given interpretation might still be the intended one.

Now, we can take the sentence "Student S1, named Anne, is enrolled on course C1" and replace each of S1, Anne, and C1 by the corresponding symbols taken from some other row in the table, such as S2, Boris, and C1. In so doing, we are applying exactly the same mode of interpretation to each row. If that is indeed how the table is meant to be interpreted, then we can conclude that the following sentences are all true:

Student S1, named Anne, is enrolled on course C1.
Student S1, named Anne, is enrolled on course C2.
Student S2, named Boris, is enrolled on course C1.
Student S3, named Cindy, is enrolled on course C3.

In Chapter 3, "Predicates and Propositions", we shall see exactly how such interpretations can be systematically formalized. In Chapter 4, "Relational Algebra—The Foundation", and Chapter 5, "Building on The Foundation", we shall see how they help us to formulate correct queries to derive useful information from a relational database.

1.5 "Collection of Variables"

Now look at Figure 1.2, a slight revision of Figure 1.1.

ENROLMENT

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

Figure 1.2: A variable, showing its current value

We have added the name, `ENROLMENT`, above the table, and we have added an extra row.

`ENROLMENT` is a *variable*. Perhaps the table we saw earlier was once its *value*. If so, it (the variable) has been *updated* since then—the row for S4 has been added. Our interpretation of Figure 1.1 now has to be revised to include the sentence represented by that additional row:

> Student S1, named Anne, is enrolled on course C1.
> Student S1, named Anne, is enrolled on course C2.
> Student S2, named Boris, is enrolled on course C1.
> Student S3, named Cindy, is enrolled on course C3.
> Student S4, named Devinder, is enrolled on course C1.

Notice that in English we can join all these sentences together to form a single sentence, using conjunctions like "and", "or", "because" and so on. If we join them using "and" in particular, we get a single sentence that is logically equivalent to the given set of sentences in the sense that it is true if *each* one of them is true (and false if *any* one of them is false). A database, then, can be thought of as a representation of an account of the enterprise expressed as a single sentence! (But it's more usual to think in terms of a collection of individual sentences.)

We might also be able to conclude that the following sentences (for example) are false:

> Student S2, named Boris, is enrolled on course C2.
> Student S2, named Beth, is enrolled on course C1.

Whenever the variable is updated, the set of true sentences represented by its value changes in some way. Updates usually reflect perceived changes in the enterprise, affecting our beliefs about it and therefore our account of it.

1.6 What Is a Relational Database?

A relational database is one whose symbols are organized into a collection of *relations*. Figure 1.3 confirms that the examples we have already seen are in fact relations, depicted in tabular form. Indeed, according to Figure 1.2, the relation depicted in Figure 1.3 is the current *value* of the variable `ENROLMENT`.

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

Figure 1.3: A relation, shown in tabular form

Happily, the visual (tabular) representation we have been using thus far is suited particularly well to relational databases: so much so that many people use the word *table* as an alternative to *relation*. The language SQL in particular uses that term, so in the context of relational theory it is convenient and judicious to stick with relation for the theoretical construct, allowing SQL's deviations from relational theory to be noted as differences between tables and relations.

Relation is a formal term in mathematics—in particular, in the logical foundation of mathematics. It appeals to the notion of relationships between things. Most mathematical texts focus on relations involving things taken in pairs but our example shows a relation involving things taken three at a time and, as we shall see, relations in general can relate any number of things (and, as we shall see, the number in question can even be less than two, making the term relation seem somewhat inappropriate).

Relational database theory is built around the concept of a relation. Our study of the theory will include:

- The "anatomy" of a relation.
- **Relational algebra:** a set of mathematical operators that operate on relations and yield relations as results.
- **Relation variables:** their creation and destruction, and operators for updating them.
- **Relational comparison operators**, allowing **consistency** rules to be expressed as **constraints** (commonly called **integrity constraints**) on the variables constituting the database.

And we will see how these, and other constructs, can form the basis of a **database language** (specifically, a *relational* database language).

1.7 "Relation" Not Equal to "Table"

"Table", here, refers to pictures of the kind shown in Figures 1.1, 1.2, and 1.3. The terms relation and table are not synonymous. For one thing, although every relation can be depicted as a table, not every table is a representation of (i.e., *denotes*) some relation. For another, several different tables can all represent the same relation. Consider Figure 1.4, for example.

Name	StudentId	CourseId
Devinder	S4	C1
Cindy	S3	C3
Anne	S1	C1
Boris	S2	C1
Anne	S1	C2

(Actually, there are two very special relations which cannot sensibly be depicted in tabular form. You will encounter these two in Chapter 4.)

Figure 1.4: Same relation as Figure 1.3

The table in Figure 1.4 is different from the one in Figure 1.3, but it represents the same relation. I have changed the order of the columns and the order of the rows, each green row in Figure 1.4 has the same symbols for each column heading as some row in Figure 1.3 and each row in Figure 1.3 has a corresponding row, derived in that way, in Figure 1.4. What I am trying to illustrate is the principle that the relation represented by a table does not depend on the order in which we place the rows or the columns in that table. It follows that several different tables can all denote the same relation, because we can simply change the left-to-right order in which the columns are shown and/or the top-to-bottom order in which the rows are shown and yet still be depicting the same relation.

What does it mean to say that the order of columns and the order of rows doesn't matter? We will find out the answer to this question when we later study the typical *operators* that are defined for operating on relations (e.g., to compute results of queries against the database) and relation variables (e.g., to update the database). None of these operators will depend on the notion of some row or some column being the first or last, or immediately before or after some other column or row.

We can also observe that not every table depicts a relation. Such tables can easily be obtained just by deleting the blue rows (the column headings) from each of Figures 1.1 to 1.4. Figure 1.5 shows another table that does not depict any relation.

A	B	A
1	2	3
4		5
6	7	8
9	9	?
1	2	3

Figure 1.5: Not a relation

The various reasons why this table cannot be depicting a relation should become apparent to you by the time you reach the end of this chapter.

1.8 Anatomy of a Relation

Figure 1.6 shows the terminology we use to refer to parts of the structure of a relation.

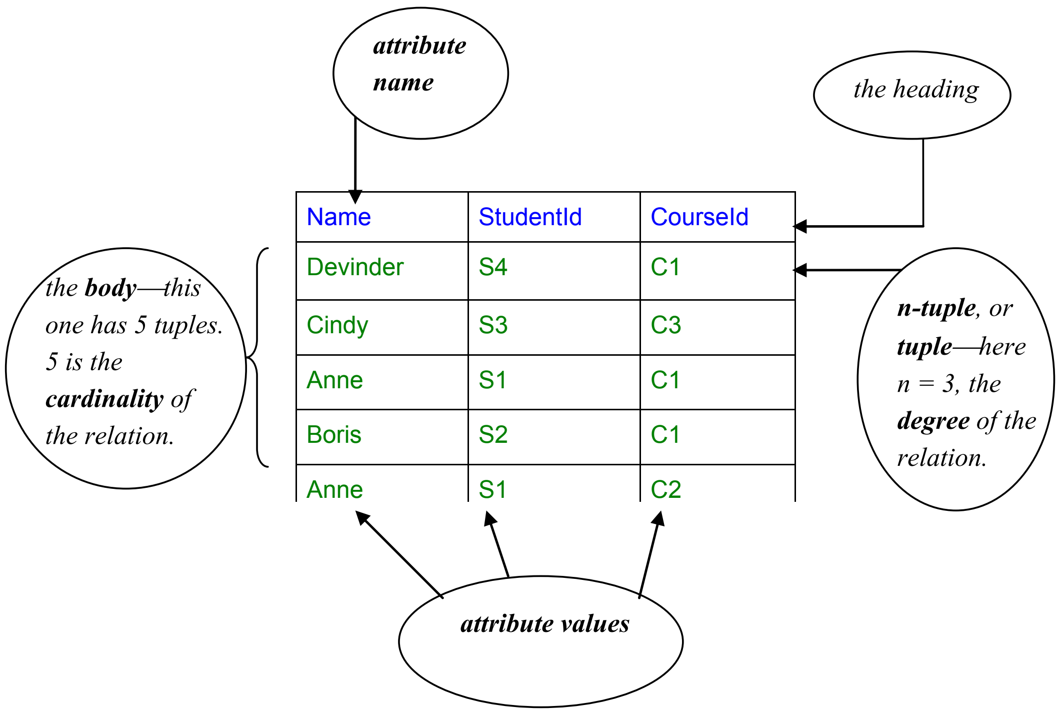

attribute name
the heading
Name
StudentId
CourseId
Devinder
S4
C1
Cindy
S3
C3
Anne
S1
C1
Boris
S2
C1
Anne
S1
C2
the body—this one has 5 tuples. 5 is the cardinality of the relation.
n-tuple, or tuple—here n = 3, the degree of the relation.
attribute values

Figure 1.6: Anatomy of a relation

Because of the distinction I have noted between the terms relation and table, we prefer not to use the terminology of tables for the anatomical parts of a relation. We use instead the terms proposed by E.F. Codd, the researcher who first proposed relational theory as a basis for database technology, in 1969.

Try to get used to these terms. You might not find them very intuitive. Their counterparts in the tabular representation might help:

• relation	:	table
• (*n*-)tuple	:	row
• attribute	:	column

Also (as shown in Figure 1.6):

The **degree** is the number of attributes.

The **cardinality** is the number of tuples.

The **heading** is the *set* of attributes (note **set**, because the attributes are not ordered in any way and no attribute appears more than once).

The **body** is the *set* of tuples (again, note **set**—the tuples are not ordered and no tuple appears more than once).

An attribute has an **attribute name**, and no two have the same name.

Each attribute has an **attribute value** in each tuple.

1.9 What Is a DBMS?

A database management system (DBMS) is exactly what its name suggests—a piece of software for managing databases and providing access to them. But be warned!—in the industry the term database is commonly used to refer to a DBMS, especially in promotional literature. You are strongly discouraged from adopting such sloppy practice (if such a system is a database, what are the things it manages?)

A DBMS responds to *commands* given by *application programs*, custom-written or general-purpose, executing on behalf of users. Commands are written in the *database language* of the DBMS (e.g., SQL). Responses include completion codes, messages and results of *queries*.

In order to support multiple concurrent users a DBMS normally operates as a *server*. Its immediate users are thus those application programs, running as *clients* of this server, typically (though not necessarily) on behalf of *end users*. Thus, some kind of *communication protocol* is needed for the transmission of commands and responses between client and server. Before submitting commands to the server a client application program must first establish a *connection* to it, thus initiating a *session*, which typically lasts until the client explicitly asks for it to be terminated. That is all you need to know about *client-server architecture* as far as this book is concerned.

This book is concerned with *relational* DBMSs and *relational* databases in particular, and soon we will be looking at the components we expect to find in a relational DBMS. Before that we need to briefly review what is expected of a DBMS in general.

1.10 What Is a Database Language?

To repeat, the commands given to a DBMS by an application are written in the database language of the DBMS. The term *data sublanguage* is sometimes used instead of database language. The sub- prefix refers to the fact that application programs are sometimes written in some more general-purpose programming language (the "host" language), in which the database language commands are embedded in some prescribed style. Sometimes the embedding style is such that the embedded statements are unrecognized by the host language compiler or interpreter, and some special *preprocessor* is used to replace the embedded statements by, for example, `CALL` statements in the host language.

A **query** is an expression that, when evaluated, yields some result derived from the database. Queries are what make databases useful. Note that a query is not of itself a command (though some texts, curiously, use the term query for commands as well as genuine queries, including commands that update the database!). The DBMS might support some kind of command to evaluate a given query and make the result available for access, also using DBMS commands, by the application program. The application program might execute such commands in order to display a query result (usually in tabular form) in a window.

1.11 What Does a DBMS Do?

In response to requests from application programs, we expect a DBMS to be able, for example, to

- create and destroy variables in the database
- take note of integrity rules (*constraints*)
- take note of *authorisations* (who is allowed to do what, to what)
- update variables (honouring constraints and authorisations)
- provide results of *queries*

To amplify some of the terms just used:

The **requests** take the form of commands written in the database language supported by the DBMS.

The **variables** are the constituents of the database, like the `ENROLMENT` variable we looked at earlier. Such variables are both *persistent* and *global.* A persistent variable is one that ceases to exist only when its destruction is explicitly requested by some user. A global variable is one that exists independently of the application programs that use it, distinguishing it from a local variable, declared within the application program and automatically destroyed when the program unit ("block") in which it is declared finishes its execution.

Constraints (sometimes called **integrity constraints**) are rules governing permissible values, and permissible combinations of values, of the variables. For example, it might be possible to tell the DBMS that no student's assessment score can be less than zero. A database that *violates* a constraint is, by definition, incorrect—it represents an account that is in some respect *false*. A database that *satisfies* all its constraints is said to be *consistent*, even though it cannot in general be guaranteed to be correct.

In the sense that constraints are for integrity, **authorisations** are for **security.** Some of the data in a database might represent sensitive information whose accessibility is restricted to certain *privileged* users only. Similarly, it might be desired to allow some users to access certain parts of the database without also being able to update those parts.

Note the three parts of an authorisation: who, what, and to what. "Who" is a user of the database; "what" is one of the operations that are available for operating on the variables in the database; "to what" is one of those variables.

In the remaining sections of this chapter you will see examples of how a relational DBMS does these things. Unless otherwise stated, the examples use commands written in **Tutorial D.**

1.12 Creating and Destroying Variables

Example 1.1 shows a command to create the variable shown in Figure 1.2:

Example 1.1: Creating a database variable.

```
VAR ENROLMENT BASE RELATION
          { StudentId  SID ,
            Name       CHAR ,
            CourseId   CID }
        KEY { StudentId, CourseId } ;
```

Explanation 1.1:

`VAR` is a key word, indicating that a variable is to be created.

`ENROLMENT` is the variable's name.

`BASE` is a key word indicating that the variable is to be part of the database, thus both persistent and global. If **`BASE`** were omitted, then the command would result in creation of a local variable.

The text from **`RELATION`** to the closing brace specifies *the declared type* of the variable, meaning that every value ever assigned to `ENROLMENT` must be a value of that type.

The declared type of `ENROLMENT` is a *relation type*, indicated by the key word `RELATION` and a *heading specification*. Thus, every value ever assigned to `ENROLMENT` must be a relation of that type. A heading specification consists of a list of attribute names, each followed by a type name, the entire list being enclosed in braces. Thus, each attribute of the heading also has a declared type. The type names `SID` and `CID` (for student ids and course ids) refer to *user-defined* types. User-defined types have to be defined by some user of the DBMS before they can be referred to. The type name `CHAR` (character strings), by contrast, is a *built-in* type: it is provided by the DBMS itself, is available to all users, and cannot be destroyed.

Chapter 2, "Values, Types, Variables, Operators", deals with types in more detail, and shows you how to define types such as `SID` and `CID`.

`KEY` indicates that the variable is subject to a certain kind of **constraint**, in this case declaring that no two tuples in the relation assigned to `ENROLMENT` can ever have the same combination of attribute values for `StudentId` and `CourseId` (i.e., we cannot enrol the same student on the same course more than once, so to speak). We will learn more about constraints in general and key constraints in particular in Chapter 6.

Destruction of `ENROLMENT` is the simple matter shown in Example 1.2,

Example 1.2: Destroying a variable.

```
DROP VAR ENROLMENT ;
```

After execution of this command the variable no longer exists and any attempt to reference it is in error.

1.13 Taking Note of Integrity Rules

For example, suppose the university has a rule to the effect that there can never be more than 20,000 enrolments altogether. Example 1.3 shows how to declare the corresponding constraint in **Tutorial D.**

Example 1.3: Declaring an integrity constraint.

```
CONSTRAINT MAX_ENROLMENTS
     COUNT ( ENROLMENT ) ⩽ 20000 ;
```

Explanation 1.3:

- **`CONSTRAINT`** is the key word indicating that a constraint is being declared.
- **`MAX_ENROLMENTS`** is the name of the constraint.
- **`COUNT ( ENROLMENT )`** is a **Tutorial D** expression yielding the cardinality (see the earlier section, "Anatomy of a Relation") of the current value of `ENROLMENT`.
- **`COUNT (ENROLMENT) ⩽ 20000`** is a truth-valued expression, yielding *true* if the cardinality is less than or equal to 20000, otherwise yielding *false*. (*Note* regarding *Rel:* Because the symbol ⩽ is normally unavailable on keyboards, *Rel* accepts <= in its place.)

The declaration tells the DBMS that the database is *inconsistent* if the value of `MAX_ENROLMENTS` is ever *false*, and that the DBMS is therefore to reject any attempt to update the database that, if accepted, would bring about that situation.

Example 1.4 shows how to retract a constraint that ceases to be applicable.

Example 1.4: Retracting an integrity constraint.

```
DROP CONSTRAINT MAX_ENROLMENTS ;
```

1.14 Taking Note of Authorisations

Tutorial D does not include any commands for creating and destroying permissions, because security and authorization, though important, are not specifically *relational* database issues. If **Tutorial D** did include such commands, we might reasonably expect them to look like those shown in Example 1.5, which are meant to be self-explanatory.

Example 1.5: Creating permissions

```
PERMISSION U9_ENROLMENT FOR User9 TO READ ENROLMENT ;
PERMISSION U8_ENROLMENT FOR User8 TO UPDATE ENROLMENT ;
```

Note the *syntactic consistency* with commands we have already seen: a key word indicating the kind of thing being created or destroyed, followed by the name of the thing, followed in turn by the specification of the thing. (C.J. Date, co-designer of **Tutorial D**, makes a slightly different suggestion for granting permissions in his *Introduction to Database Systems*, 8th edition, on page 506.)

How do you rate computer languages you are already acquainted with, for syntactic consistency? For example, the database language SQL has been noted to suffer from several syntactic *inconsistencies* (as well as—much more seriously—several harmful deviations from relational database theory).

By now you can predict the command, consistent with Example 1.5 and shown in Example 1.6, to be used to retract a permission previously granted.

Example 1.6: Retracting a permission

```
DROP PERMISSION U9_ENROLMENT ;
```

In case you are familiar with SQL's `GRANT` and `REVOKE` statements that are used for such purposes, you might like to give some thought to the advantages and disadvantages of using specific names for permissions. SQL doesn't use them—in an SQL `REVOKE` statement you have to repeat the details of the permission you are withdrawing.

1.15 Updating Variables

The usual way of updating a variable in computer languages is by *assignment*. For example, if `X` is an integer variable, the assignment `X := X + 1` updates `X` such that its value immediately after execution of the assignment is one more than its value was immediately beforehand. The expression on the right of `:=` denotes the *source* for the assignment and the variable name on the left denotes the *target*.

When the target is a relation variable—as it always is when it is part of a relational database—the source must be a relation. You will learn how to write expressions that denote relations in Chapters 2, 4 and 5, but in any case assignment, though it *should* be available (it isn't in SQL), is not the usual way of applying updates to a relational database. This is because there is very often only a small amount of difference, in a manner of speaking, between the "old" value and the "new" value and it is usually much more convenient to be able to express the update in terms of that small difference.

The differential update operators expected in a relational DBMS are usually called `INSERT`, `DELETE`, and `UPDATE`, and those are the names used in **Tutorial D** (also in SQL). Take a look at `DELETE` first (Example 1.8).

Example 1.8: Updating by deletion

```
DELETE ENROLMENT WHERE StudentId = SID ( 'S4' ) ;
```

Explanation 1.8:

- Informally, Example 1.8 deletes all the tuples for student S4 and can be interpreted as meaning "student S4 is no longer enrolled on any courses". More formally, it assigns to the variable `ENROLMENT` the relation whose body consists of those tuples in the current value of `ENROLMENT` that fail to satisfy the condition given in the `WHERE` clause—thus, every tuple in which the value of the StudentId attribute is *not* the student identifier S4.
- **`StudentId = SID ( 'S4' )`** is a conditional expression. Because it follows the key word `WHERE` here, it is in fact a `WHERE` condition, also known as a *restriction condition*.
- The expression `SID ( 'S4' )` will be explained in Chapter 2, when we study *types*.

Next, in Example 1.9, we look at `UPDATE`.

Example 1.9: Updating by replacement

```
UPDATE ENROLMENT WHERE StudentId = SID ( 'S1' ) :
       { Name := 'Ann' } ;
```

Note that `UPDATE` uses a `WHERE` clause, just like `DELETE`. The `WHERE` clause is followed by a list of assignments—in Example 1.9 just one assignment—but these are assignments to attributes, not assignments to variables.

Explanation 1.9:

- Informally, Example 1.9 updates each `ENROLMENT` tuple for student S1, changing its `Name` value to `'Ann'`. More formally, it assigns to the variable `ENROLMENT` the relation that is identical to the current value in all respects except that the value for the attribute `Name`, in the tuples whose `StudentId` value is the student identifier S1, becomes the string `'Ann'` in each case. (I would have written "except possibly" had I not known that the existing `Name` value in those tuples is `'Anne'` in each case. In some circumstances no change takes place as a result of executing an `UPDATE`, and the same applies to `DELETE` and `INSERT`.)
- **`Name := 'Ann'`** is an *attribute assignment.* An attribute assignment sets the value of the target attribute to the specified value, in each tuple that satisfies the `WHERE` condition.

Finally, Example 1.10 illustrates the use of `INSERT`.

Example 1.10: Updating by insertion

```
INSERT ENROLMENT
  RELATION {
    TUPLE { StudentId SID ( 'S4' ) ,
      Name 'Devinder' ,
      CourseId CID ( 'C1' ) } } ;
```

Explanation 1.10:

- Informally, Example 1.10 adds a tuple to `ENROLMENT` indicating that student S4, still called Devinder, is now enrolled on course C1. More formally, it assigns to the variable `ENROLMENT` the relation consisting of every tuple in the current value of `ENROLMENT` and every tuple (there is only one in this particular example) in the relation denoted by the expression following the word `ENROLMENT`.
- The expression beginning with the key word `TUPLE` and ending at the penultimate closing brace denotes the tuple consisting of the three indicated attribute values: `SID ( 'S4' )` for the attribute `StudentId`, `'Devinder'` for the attribute `Name`, and `CID ( 'C1' )` for the attribute `CourseId`.
- The expression beginning with the key word `RELATION` and ending at the final closing brace denotes the relation whose body consists of that single tuple. Such expressions are fully explained in Chapter 2, "Values, Types, Variables, Operators".

Example 1.8 has no effect on the database in the case where the current value of `ENROLMENT` has no tuples for student S4.

Example 1.9 has no effect on the database in the case where the current value of `ENROLMENT` has no tuples for student S1.

Example 1.10 has no effect on the database in the case where the current value of `ENROLMENT` already contains the tuple representing the enrolment of student S4, named Devinder, on course C1. It also has no effect on the database if the cardinality of the current value of `ENROLMENT` is 20,000 and the constraint `MAX_ENROLMENTS` (Example 1.3) is in effect. In this case, and possibly in the first case too, an error message results.

1.16 Providing Results of Queries

Expressing queries in **Tutorial D** is the (big) subject of Chapters 4 and 5. Here I present just a simple example to give you the flavour of things to come in those chapters. Example 1.11 is a query expressing the question, who is enrolled on course C1?

Example 1.11: A query in Tutorial D

```
ENROLMENT WHERE CourseId = CID('C1')
    { StudentId, Name }
```

Note carefully that Example 1.11 is not a command. It is just an expression, denoting a value—in this case, a relation. In a relational database language the result of a query is always another relation! Figure 1.7 shows the result of Example 1.11 in the usual tabular form.

StudentId	Name
S1	Anne
S2	Boris
S4	Devinder

Figure 1.7: Result of query in Example 1.11

Explanation 1.11:

- `WHERE` is the key word identifying the **Tutorial D** operator of that name. This operator operates on a given relation and yields a relation. Certain operators, including this one, that operate on relations and yield relations together constitute the *relational algebra*, covered in detail in Chapter 4.
- `CourseId = CID('C1')` qualifies `WHERE`, specifying that just the tuples for course C1 are required.
- `{ StudentId, Name }` specifies that from the result of the previous operation (`WHERE`) just the `StudentId` and `Name` attributes are required.

The overall result is a relation formed from the current value of `ENROLMENT` by discarding certain tuples and a certain attribute.

EXERCISE

Consider the table shown in Figure 1.5, repeated here for convenience:

A	B	A
1	2	3
4		5
6	7	8
9	9	?
1	2	3

Give three reasons why it cannot possibly represent a relation. By the way, this table is supported by SQL, and the three reasons represent some of SQL's serious and far-reaching deviations from relational theory.

2 Values, Types, Variables, Operators

2.1 Introduction

In this chapter we look at the four fundamental concepts on which most computer languages are based. We acquire some useful terminology to help us talk about these concepts in a precise way, and we begin to see how the concepts apply to relational database languages in particular. It is quite possible that you are already very familiar with these concepts—indeed, if you have done any computer programming they cannot be totally new to you—but I urge you to study the chapter carefully anyway, as not everybody uses exactly the same terminology (and not everybody is as careful about their use of terminology as we need to be in the present context). And in any case I also define some special terms, introduced by C.J. Date and myself in the 1990s, which have perhaps not yet achieved wide usage—for example, *selector* and *possrep*.

I wrote "most computer languages" because some languages dispense with variables. Database languages typically do not dispense with variables because it seems to be the very nature of what we call a database that it varies over time in keeping with changes in the enterprise. Money changes hands, employees come and go, get salary rises, change jobs, and so on. A language that supports variables is said to be an imperative language (and one that does not is a functional language). The term "imperative" appeals to the notion of *commands* that such a language needs for purposes such as updating variables. A command is an instruction, written in some computer language, to tell the system to *do* something. The terms *statement* (very commonly) and *imperative* (rarely) are used instead of *command*. In this book I use *statement* quite frequently, bowing to common usage, but I really prefer *command* because it is more appropriate; also, in normal discourse *statement* refers to a sentence of the very important kind described in Chapter 3 and does not instruct anybody to do anything.

2.2 Anatomy of A Command

Figure 2.1 shows a simple command—the assignment, `Y := X + 1`—dissected into its component parts. The annotations show the terms we use for those components.

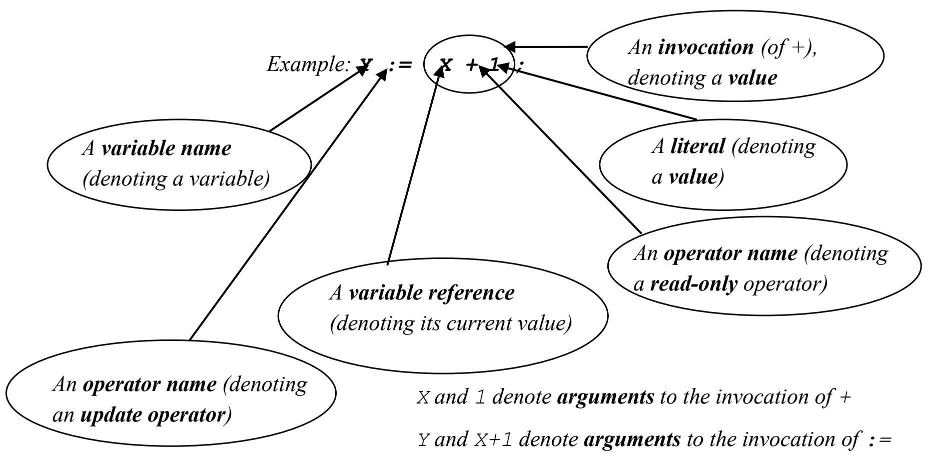

Example: Y := X + 1 ;
An invocation (of +), denoting a value
A variable name (denoting a variable)
A literal (denoting a value)
An operator name (denoting a read-only operator)
A variable reference (denoting its current value)
An operator name (denoting an update operator)
X and 1 denote arguments to the invocation of +
Y and X+1 denote arguments to the invocation of :=

Figure 2.1: Some terminology

It is important to distinguish carefully between the concepts and the language constructs that represent (*denote*) those concepts. It is the distinction between what is written and what it means—*syntax* and *semantics*.

Each annotated component in Figure 1 is an example of a certain language construct. The annotation shows the term used for the language construct and also the term for the concept it denotes. Honouring this distinction at all times can lead to laborious prose. Furthermore, we don't always have distinct terms for the language construct and the corresponding concept. For example, there is no single-word term for an expression denoting an *argument*. We can write "argument expression" when we need to be absolutely clear and there is any danger of ambiguity, but normally we would just say, for example, that `X+1` *is* an argument to that invocation of the operator "`:=`" shown in Figure 2.1. (The real argument is the result of evaluating `X+1`.)

The update operator "`:=`" is known as assignment. The command `Y := X+1` is an invocation of assignment, often referred to as just an assignment. The effect of that assignment is to evaluate the expression `X+1`, yielding some numerical result *r* and then to assign *r* to the variable `Y`. Subsequent references to `Y` therefore yield *r* (until some command is given to assign something else to `Y`).

Note the two operands of the assignment: `Y` is the *target*, `X+1` the *source*. The terms *target* and *source* here are names for the *parameters* of the operator. In the example, the argument expression `Y` is substituted for the parameter *target* and the argument expression `X+1` is substituted for the parameter *source*. We say that *target* is subject to update, meaning that any argument expression substituted for it must denote a variable. The other parameter, *source*, is not subject to update, so any argument expression substituted must denote a value, not a variable. `Y` denotes a variable and `X+1` denotes a value. When the assignment is evaluated (or, as we sometimes say of commands, executed), the variable denoted by `Y` becomes the argument substituted for *target*, and the current value of `X+1` becomes the argument substituted for *source*.

Whereas the `Y` in `Y := X + 1` denotes a variable, as I have explained, the `X` in `Y := X + 1` does not, as I am about to explain. So now let's analyse the expression `X+1`. It is an invocation of the read-only operator `+`, which has two parameters, perhaps named *a* and *b*. Neither *a* nor *b* is subject to update. A read-only operator is one that has no parameter that is subject to update. Evaluation of an invocation of a read-only operator yields a value and updates nothing. The arguments to the invocation, in this example denoted by the expressions `X` and `1`, are the values denoted by those two expressions. `1` is a *literal*, denoting the numerical value that it always denotes; `X` is a *variable reference*, denoting the value currently assigned to `X`.

A literal is an expression that denotes a value and does not contain any variable references. But we do not use that term for all such expressions: for example, the expression `1+2`, denoting the number 3, is not a literal. I defer a precise definition of *literal* to later in the present chapter.

2.3 Important Distinctions

The following very important distinctions emerge from the previous section and should be firmly taken on board:

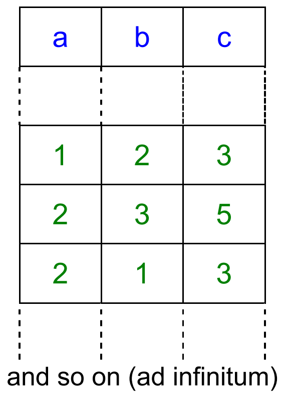
- Syntax versus semantics
- Value versus variable
- Variable versus variable reference
- Update operator versus read-only operator
- Operator versus invocation
- Parameter versus argument
- Parameter subject to update versus parameter not subject to update

Each of these distinctions is illustrated in Figure 2.1, as follows:

- **Value versus variable:** `Y` denotes a variable, `X` denotes the value currently assigned to the variable `X`. `1` denotes a value. Although `X` and `Y` are both symbols referencing variables, what they denote depends in the context in which those references appear. `Y` appears as an update target and thus denotes the variable of that name, whereas `X` appears where an expression denoting a value is expected and that position denotes the current value of the referenced variable. Note that variables, by definition, are subject to change (in value) from time to time. A value, by contrast, exists independently of time and space and is not subject to change.
- **Update operator versus read-only operator:** ":=" (assignment) is an update operator; "+" (addition) is a read-only operator. An update operator has at least one parameter that is subject to update; a read-only operator doesn't. A read-only operator, when invoked, yields a value; an update operator doesn't.
- **Operator versus invocation:** "+" is an operator; the expression `X+1` denotes an invocation of "+".
- **Parameter versus argument:** The expressions `X` and `1` denote arguments to the invocation of +; the operator + is defined to have two parameters. When an operator is invoked, an argument must be substituted for each of its defined parameters. The term *argument* strictly refers to the value or variable denoted by the argument expression but is often used to refer to the expression itself.
- **Parameter subject to update versus parameter not subject to update:** The first parameter of ":=" (the one representing the target) is subject to update, so a variable must be substituted for it when ":=" is invoked (and an expression denoting a variable must appear in the corresponding position in the expression denoting the invocation); the second parameter of ":=" and both parameters of + are not subject to update, so values must be substituted for them in invocations (and expressions denoting values must appear in the corresponding positions in the expressions denoting the invocations).

2.4 A Closer Look at a Read-Only Operator (+)

A read-only operator is what mathematicians call a *function*, and a function turns out to be just a special case of a relation! Because it is a relation, a function can be depicted in tabular form. Figure 2.2 is a picture of part of the function represented by the read-only operator +.



a	b	c
1	2	3
2	3	5
2	1	3

and so on (ad infinitum)

Figure 2.2: The operator + as a relation (part)

The relation shown in Figure 2.2 represents the predicate *a* + *b* = *c*. The relation attributes *a* and *b* can be considered as the parameters of the operator +. Each tuple maps a pair of values substituted for *a* and *b* to the result of their addition, which is substituted for *c*. The relation is a function because each unique <*a*,*b*> pair maps to exactly one *c* value—no two tuples with the same *a* value also have the same *b* value, so, given an *a* and a *b*, so to speak, we know the (only) resulting *c*.

Notice how the relational perception of an operator neutralises the distinction between arguments and result. This relation could also represent the predicate *c*—*b* = *a*, or *c*—*a* = *b*.

You can imagine the invocation `1+2` as singling out the tuple with *a*=1 and *b*=2 (there is only one such tuple) and yielding the *c* value (3) in that tuple.

This particular relation is concerned only with numbers—its *domain of discourse*, some would say. Mathematicians, perceiving + as a function mapping pairs of numbers (<*a*,*b*>) to numbers (*c*), call the (<*a*,*b*>) number-pairs the *domain* of the function and the numbers (*c*) its *range*. Computer scientists, perceiving + as an operator, say that its parameters *a* and *b* are of type number, as is the result, *c* (the type of the result is also referred to as the type of the operator).

2.5 Read-only Operators in **Tutorial D**

In computer languages we distinguish between operators that are defined as part of the language and operators that may be defined by uses of the language. Those defined as part of the language are called *built-in,* or *system-defined,* operators whereas those defined by users are called *user-defined* operators.

A complete grammar for **Tutorial D** does not yet exist and the incomplete one that does exist does not give a complete list of built-in operators. It mentions a few particular ones that have been devised for certain special purposes and adds "…plus the usual possibilities", leaving it to the implementation to decide what the usual possibilities are. In this book the matter of whether an operator used in my examples is built-in or user-defined is immaterial, except of course for those operators which an implementation is explicitly required to provide as built-in.

User-defined operator definition in **Tutorial D** is illustrated in Example 2.1, which defines an operator named `HIGHER_OF` to give the value of whichever is the higher of two given integers. For example, the invocation `HIGHER_OF(3,4)` yields the integer 4.

Example 2.1: A User-Defined Operator

```
OPERATOR HIGHER_OF ( A INTEGER, B INTEGER ) RETURNS INTEGER ;
IF A > B THEN RETURN A ;
        ELSE RETURN B ;
END IF ;
END OPERATOR ;
```

Explanation 2.1:

- **`OPERATOR HIGHER_OF`** announces that an operator is being defined and its name is `HIGHER_OF`. There might be other operators, also named `HIGHER_OF`, in which case they are distinguished from one another by the types of their parameters. The name combined with the parameter definitions is called the *signature* of the operator. Here the signature is `HIGHER_OF(A INTEGER,B INTEGER)`, which would distinguish it from `HIGHER_OF(A RATIONAL,B RATIONAL)` if that operator were also defined.
- **`A INTEGER, B INTEGER`** specifies two parameters, named `A` and `B` and both of declared type `INTEGER`. (Although I have included parameter names in the signature, they do not normally have any significance in distinguishing one operator from another. That is because parameter names are not normally used in invocations, the connections between argument expressions and their corresponding parameters being established by position rather than by use of names.)
- **`RETURNS INTEGER`** specifies that the value resulting from every invocation of `HIGHER_OF` shall be of type `INTEGER` (which is thus the declared type of `HIGHER_OF`).
- **`IF`** … **`END IF ;`** is a single command (specifically, an `IF` statement) constituting the program code that implements `HIGHER_OF`. The programming language part of **Tutorial D**, intended for writing implementation code for operators and applications, is really beyond the scope of this book, but if you are reasonably familiar with programming languages in general you should have no trouble understanding **Tutorial D**, which is deliberately both simple and conventional.
  The `IF` statement contains further commands within itself…
- **`IF A > B THEN RETURN A`** … such as `RETURN A` here, which is executed only when the given `IF` condition, `A > B`, evaluates to `TRUE` (i.e., is satisfied by the arguments substituted for `A` and `B` in an invocation of `HIGHER_OF`). The `RETURN` statement terminates the execution of an invocation and causes the result of evaluating the given expression, `A`, to be the result of the invocation.
- **`ELSE RETURN B`** specifies the statement to be executed when the given `IF` condition is not satisfied.
- **`END IF`** marks the end of the `IF` statement.
- **`END OPERATOR`** marks the end of the program code and in fact the end of the operator definition.

Notes concerning *Rel*:

- *Rel* provides as built-in all the **Tutorial D** operators used in this book except where explicitly stated to the contrary.
- *Rel* supports **Tutorial D** user-defined operators.
- *Rel* additionally supports user-defined operators with program code written in Java™ (the language in which *Rel* itself is implemented), indicated by the key word `FOREIGN`. Examples of such operators are provided in the download package for *Rel*. Here are two of them (as provided at the time of writing in Version 3.15, in the file `OperatorsChar.d`, which you can load and execute in Dbrowser):

```
OPERATOR SUBSTRING(s CHAR, beginindex INTEGER, endindex
INTEGER) RETURNS CHAR Java FOREIGN
// Substring, 0 based
return new ValueCharacter(s.stringValue().substring(
                                (int)beginindex.longValue(),
                                (int)endindex.longValue()));
END OPERATOR;

OPERATOR SUBSTRING(s CHAR, index INTEGER) RETURNS CHAR Java
FOREIGN
// Substring, 0 based
return new ValueCharacter(s.stringValue().substring(
                                   (int)index.longValue()));
END OPERATOR;
```

Notice that these two operators are both named `SUBSTRING`, the first having three parameters, the second two. Thus, *Rel* can tell which one is being invoked by a particular expression of the form `SUBSTRING( … )` according to the number of arguments to the invocation (and in fact according to the declared types of the expressions denoting those arguments). The first, when invoked, yields the string that starts at the given `beginindex` position within the given string `s`, and ends at the given `endindex` position, where 0 is the position of the first character in `s`. The second yields the string that starts at the given `index` position in `s` and ends at the end of `s`. Hence, `SUBSTRING('database',2,4) = 'tab'` and `SUBSTRING('database',4) = 'base'`.

I do not offer an explanation of the Java™ code used in these examples, that being beyond the scope of this book.

2.6 What Is a Type?

A type is a *named* set of values. Much of the relational database literature, especially the earlier literature, uses the term *domain* for this concept, because that was the term E.F. Codd used. Nowadays we prefer *type* because that is the term most commonly used for the concept in computer science. Codd's term *domain* derived from the fact that he used it exclusively to refer to the *declared type* of an attribute of a relation.

For example, there might be a type named `WEEKDAY` whose values constitute the set `{ Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }`. For another example, type `INTEGER` is commonly available in computer languages, its values being all the integers in some range, such as $-(2^{32})$ to $2^{32}-1$. Terms such as `Monday` and `-1` are *literals*. `Monday` denotes a certain value of type `WEEKDAY` and `-7` denotes a certain value of type `INTEGER`. It is essential that every value that can be operated on in a computer language can be denoted by some literal in that language.

In any computer language that supports types (as most of them do), some types are *built-in* (provided as part of the language). In some languages the only supported types are the built-in ones, but the trend in modern languages has been towards inclusion of support for *user-defined types* too.

The built-in types of **Tutorial D** are:

- `CHARACTER` or, synonymously, `CHAR`, for character strings.
- `INTEGER` or, synonymously, `INT`, for integers.
- `RATIONAL` for rational numbers, denoted by numeric literals that include a decimal point, such as `3.25, 1.0, 0.0, -7.935`.
- `TUPLE` types and `RELATION` types as described later in this Chapter.

2.7 What Is a Type Used For?

In general, a type is used for *constraining* the values that are permitted to be used for some purpose. In particular, for constraining:

- the values that can be assigned to a variable
- the values that can be substituted for a parameter
- the values that an operator can yield when invoked
- the values that can appear for a given attribute of a relation

In each of the above cases, the type used for the purpose in question is the *declared type* of the variable, parameter, operator, or attribute, respectively. As a consequence, every expression denoting a value has a declared type too, whether that expression be a literal, a reference to a variable, parameter, or attribute or an invocation of an operator. Most importantly, these uses for types enable a processor such as a compiler or a DBMS to detect errors at "compile-time"—by mere inspection of a given script—that would otherwise arise, and cause much more inconvenience, at run-time (when the script is executed). Thus, support for types is deemed essential for the development of robust application programs.

2.8 The Type of a Relation

Now, if every value is of some type, and a relation, as we have previously observed, is a value, then we need to understand to what type a given relation belongs, and we need a name for that type. Here again (Figure 2.3) is our running example of a relation:

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

Figure 2.3: Enrolments again

At this stage it is perhaps tempting to conclude that relations are all of the same type, which we might as well call RELATION, just as all integers are of type INTEGER. However, it turns out to be much more appropriate, as we shall see, to consider relations of the same heading to be of the same type and relations of different headings to be of different types. But all the types whose values are relations have that very fact—that their values are relations—in common, and we call them *relation types*.

If relation types are distinguished by their headings, it is clear that a relation type name must include a specification of its heading. In **Tutorial D**, therefore, the type name for the relation shown in Figure 2.3 can be written as

```
RELATION { StudentId SID, Name NAME, CourseId CID }
```

or, equivalently (for recall that there is no ordering to the elements of a set),

```
RELATION { Name NAME, StudentId SID, CourseId CID }
```

(and so on). Note the braces, { }. **Tutorial D** *always* uses braces to enclose an enumeration of the elements of set. In fact, { StudentId SID, Name NAME, CourseId CID } denotes a set of attributes, each consisting of an attribute name followed by a type name. SID is the declared type of the attribute StudentId , NAME that of Name, and CID that of CourseId.

RELATION { StudentId SID, Name NAME, CourseId CID } might in fact be the declared type of a relation variable ENROLMENT, and that variable might be part of some database.

Clearly, there is in theory an infinite number of relation types, because there is no limit to the degree of a relation. (Recall that the degree of a relation is the number of its attributes.)

Here are some more relation types:

```
RELATION { StudentId SID, CourseId CID }
RELATION { a INTEGER, b INTEGER, c INTEGER }
RELATION { n INTEGER, w WEEKDAY }
RELATION { }
```

That last one looks a bit special! It does indeed merit special attention. We will have more to say about it later.

Relation types have other things in common about them, even though they are different types. We can see already, for example, that every relation type is defined by a set of attributes (the empty set in one particular case). Of particular interest are the *read-only operators* defined for operating on relations of all types. These operators are described in Chapter 4.

2.9 Relation Literals

In Section **2.6, What Is a Type?**, we noted the essential need to be able to denote every value that can be operated on in a computer language by some literal in that language. We have also noted that a relation, such as the one shown in Figure 2.3, is a value. What might a literal look like that denotes that value?

Well, we might try something like what is shown in Example 2.2, in which the key word `RELATION` is followed by a list of expressions denoting tuples—each one a putative *tuple literal*, in fact—enclosed in braces.

Example 2.2: A Relation Literal (not good enough!)

```
RELATION {
  TUPLE { StudentId S1, CourseId C1, Name Anne  },
  TUPLE { StudentId S1, CourseId C2, Name Anne  },
  TUPLE { StudentId S2, CourseId C1, Name Boris },
  TUPLE { StudentId S3, CourseId C3, Name Cindy },
  TUPLE { StudentId S4, CourseId C1, Name Devinder}
  }
```

But of course it is not reasonable to expect a computer language to recognise symbols such as `S1,C1,` and `Boris`. We need a proper way of writing literals for those student identifiers, course identifiers, and names.

Recall the declared types of the attributes: `SID, NAME, CID`. These are necessarily user-defined types, for it would not be reasonable to expect them to be built-in.

Suppose that values of type `SID` *are represented by* character strings (values of type `CHAR`). `CHAR` might well be a built-in type, and is indeed built-in in **Tutorial D** and *Rel*. Suppose further that character strings are denoted by text in quotes, like this: `'S1'`, as indeed they are in most computer languages. Then a literal for the student identifier S1 might be: `SID ( 'S1' )`. This literal is an invocation of the operator whose name happens to be the same as that of the type for student identifiers, `SID`. This operator has a single parameter whose declared type, `CHAR`, is that of the representation (character strings) chosen for student identifiers. In this invocation the `CHAR` literal `'S1'` denotes the argument corresponding to that parameter. The result of the invocation is not a character string but a value of type `SID`.

We call the operator `SID` a *selector,* because it can be used to "select" *any* value of type `SID`. Now, it is very likely that not every character string can validly represent a student identifier. Perhaps student identifiers must each be the letter S, followed by a maximum of four numeric digits. In that case we can expect the operator `SID`, when it is invoked, to check that the given string conforms to this rule—and raise an error if it doesn't. That is one good reason why type `SID` might be chosen in preference to type `CHAR` for student identifiers.

By the way, you can think of the literal `'S1'` as "selecting" a `CHAR` value. Every `CHAR` value can be denoted by a sequence of characters enclosed in quotes, and every sequence of characters enclosed in quotes does denote a `CHAR` value; so this syntax for literals does satisfy the requirements for being a selector.

In Example 2.2 we tried to write a relation literal by specifying a set of tuple literals, and we tried to write a tuple literal by specifying a set of attribute values. Now that you know how to specify those attribute values properly, you can easily see that the correct way of writing the first of those tuple literals, arising from the foregoing discussion, is like this:

```
TUPLE { StudentId SID('S1'), CourseId CID('C1'),
        Name NAME('Anne') }
```

The literals for course identifiers and names assume that those, too, are represented by character strings. Perhaps a course identifier must be the letter "C" followed by numeric digits, and perhaps a name must be an upper-case letter followed by lower-case letters, possibly with imbedded hyphens or spaces.

From that tuple literal it is easy to see how to write the complete relation literal (Example 2.3) in **Tutorial D**.

Example 2.3: A Relation Literal (correct version)

```
RELATION {
   TUPLE { StudentId SID('S1'), CourseId CID('C1'),
           Name NAME('Anne')},
   TUPLE { StudentId SID('S1'), CourseId CID('C2'),
           Name NAME('Anne')},
   TUPLE { StudentId SID('S2'), CourseId CID('C1'),
           Name NAME('Boris')},
   TUPLE { StudentId SID('S3'), CourseId CID('C3'),
           Name NAME('Cindy')},
   TUPLE { StudentId SID('S4'), CourseId CID('C1'),
           Name NAME('Devinder')}
   }
```

Note that the relation literal given in Example 2.3 is an invocation of a certain *relation selector*—the specific selector for relations of that specific type. Similarly, a tuple literal is an invocation of a certain *tuple selector.* An invocation of a selector is not necessarily a literal. The relation selector takes a list of *tuple expressions*, each of which *might* be a literal but is not required to be one. Similarly, the tuple selector takes a list of attribute name/attribute value pairs, each of which *might* use a literal for the value but is not required to use one—any expression of the same type as the declared type of the attribute can be used.

A note concerning abbreviations in Tutorial D: The abbreviations `REL` and `TUP` can be used in place of the key words `RELATION` and `TUPLE`. Other abbreviations such as `INT` for `INTEGER`, `CHAR` for `CHARACTER`, and `BOOL` for `BOOLEAN` are also available

Now I can return to a small matter I deferred from Section 2.2, the definition of *literal*. As already stated, a literal is an expression that contains no variable references. We can now add that it is, specifically, an invocation of a *selector*. The invocation can be explicit, as in the relation and tuple literals we have looked at, and as in the `SID`, `CID`, and `NAME` literals contained in those tuple literals. It can also be *implicit*, as in the case of, for example, `'Cindy'`, denoting a value of the system-defined type `CHAR` and 1, denoting a value of the system-defined type `INTEGER`. Because these types are system-defined, the language can include special syntax for invoking their selectors, for which it doesn't necessarily need to provide names.

CHAR literals in *Rel*:

Rel allows `CHAR` literals to be enclosed either in quotes, as already shown, or in double-quotes, like this: "`S1`". Thus, "`S1`" and '`S1`' both denote the same CHAR value. This makes it easier to write `CHAR` literals for values that include quotes. **Tutorial D** officially uses only single quotes for `CHAR` literals.

2.10 Types and Representations

According to the authors of reference [4]:

> "A major purpose of type systems is to avoid embarrassing questions about representations, and to forbid situations in which these questions might come up."

Consider again the invocation `SID('S1')`, a literal of type `SID`. Recall that `SID` is an operator that, when invoked with a suitable character string, returns a value of type `SID`; also that every value of type `SID` can be denoted by some invocation of the operator `SID`. I have explained that we call such an operator a *selector* (for values of the type in question).

The parameters of a selector correspond to components of what we call a *possible representation* (*possrep* for short). I will explain why we use the word "possible" here in just a moment. First, I want to show how a type is defined in **Tutorial D**, using a single command. In the case of type `SID`, whose values are "possibly represented" by simple character strings, the possrep consists of a single component of type `CHAR`. I have also suggested that a character string representing a student identifier might have to be of a particular format, such as the letter "S" followed by a digit sequence of limited length; and that the format in question would be enforced by the selector. The **Tutorial D** type definition shown in Example 2.4 specifies the appropriate possrep and expresses the suggested format as a *constraint*.

Example 2.4: A Type Definition

```
TYPE SID POSSREP SID { C CHAR
                       CONSTRAINT LENGTH(C) <= 5
                                  AND
                                  STARTS_WITH(C, 'S')
                                  AND
                                  IS_DIGITS(SUBSTRING(C,1)))
                     }
        INIT SID('S1');
```

Explanation 2.4:

- **`TYPE SID`** announces that a type named `SID` is being defined to the system.
- **`POSSREP SID`** announces that what follows, in braces, specifies a possible representation for values of the type. It means that the operators defined for type `SID` behave as if values of type `SID` were represented that way, regardless of how they are *physically* represented "under the covers". That is why we use the word "possible"—the values might possibly be represented internally that way (but they don't have to be and we don't even know if they are). In this case the name of that possrep is the same as the type name, `SID` (as it would be if we omitted the name). (**Tutorial D** allows more than one possrep to be given for the same type, but such complications are beyond the scope of this book. I do not deal with types in any depth. It is sufficient for present purposes just to understand how they exist, what they are for, and how to use them.)
- **`C CHAR`** defines the first and only component of the possrep, naming it `C` and specifying `CHAR` as its declared type. This definition causes an operator, `THE_C`, to come into existence. `THE_C` takes a value, *s*, of type `SID` and returns the value of the `C` component of *s* under the possible representation `SID`.
- **`CONSTRAINT`** announces that the expression following it (up to but excluding the closing brace) is a condition that must be satisfied by all possrep values that do indeed represent values of type `SID`. Note that the expression itself uses the logical connective `AND`, with its usual meaning, to connect three expressions, two of which are *comparisons* and each of which is a *truth-valued* expression—one that, when evaluated, yields either `TRUE` or `FALSE`.

  The operators `LENGTH, STARTS_WITH, SUBSTRING`, and `IS_DIGITS,` invoked in the constraint expression, are not defined as built-in operators in **Tutorial D**. I am assuming their existence as user-defined operators. Happily, their definitions are contained in the file `OperatorsChar.d` included in the download package for *Rel*.
- **`LENGTH(C) <= 5`** expresses a rule to the effect that the total length of a value for the `C` possrep component must never exceed 5.

- **STARTS_WITH(C, 'S')** returns TRUE if and only if the string given as the first operand starts with the string given as the second operand—in this case the string consisting of just the capital letter 'S'.
- **SUBSTRING(C,1)** denotes the string consisting of the whole of the value of the C possrep component apart from the first character. This is given as the argument to an invocation of IS_DIGITS, which takes a string and yields TRUE if every character in the given string is a numeric digit, otherwise FALSE.
- **INIT SID('S1')** specifies an *example value* for the type, just to confirm that values exist for the type being defined.

2.11 What Is a Variable?

Example 2.5 shows a variable declaration in **Tutorial D**.

Example 2.5: A Variable Declaration

```
VAR SN SID INIT SID ( 'S1' ) ;
```

Note that a variable is declared by means of a VAR statement.

Explanation 2.5:

- **VAR SN** announces that what follows defines a variable named SN.
- **SID**, immediately following SN, specifies the declared type of this variable, indicating that only values of type SID can be assigned to SN.
- **INIT** specifies that what follows is an expression whose value is to be immediately assigned to SN. You already know what SID ('S1') denotes (see Section 2.9). The value specified in an INIT clause is commonly called the *initial value* of the variable in question. It remains that value until it is replaced by subsequent invocation of an update operator such as assignment.

A variable is a container or holder for a value. To answer the question posed in the section heading, we can now see that a variable:

- has a name;
- has a declared type, namely the type of the values it can legally hold;
- always "has" a value (of its declared type).

The name and declared type remain the same throughout the existence of the variable, but its value can change from time to time, which is of course why it is called a variable. Although the value can change from time to time, the value must always be a value of the declared type of the variable—in Example 2.5 a value of type `SID`.

The variable `SN` is an example of a *scalar* variable in **Tutorial D**. Such variables are for use only as local variables in computer programs, such as an application program or the code implementing a user-defined operator. Of more interest in the context of relational databases are *relation variables,* known for short and from here onwards as *relvars*. Unlike scalar variables, relvars can appear as persistent objects in the database—in fact, these are the only variables permitted as persistent objects in the database. Example 2.6 shows a relvar declaration in **Tutorial D**.

Example 2.6: A Relvar Declaration

```
VAR ENROLMENT BASE RELATION { StudentId SID,
                              Name NAME,
                              CourseId CID }
                        KEY { StudentId, CourseId } ;
```

Explanation 2.6:

- **`VAR ENROLMENT`** announces that what follows defines a variable named `ENROLMENT`.
- **`BASE`**, immediately following `ENROLMENT`, specifies that the variable is to be a persistent object in the database.
- **`RELATION { … }`** specifies the declared type of the variable.
- **`KEY { StudentId, CourseId }`** specifies a certain very important kind of constraint that **Tutorial D** requires to be specified for all base relvars. Keys are covered in detail in Chapter 5. This one specifies that at no time can two distinct tuples appear in the current value of `ENROLMENT`, having the same values for `StudentId` and also the same values for `CourseId`. In enterprise terms, no two enrolments can involve the same student and the same course.

No `INIT` specification is given in Example 2.6. In the absence of an `INIT` specification in a **Tutorial D** relvar declaration the relvar is initialised to the empty relation of its declared type.

2.12 Updating a Variable

A value is assigned to a variable by invoking some *update operator*. The simplest and most general of such operators is the assignment operator itself—"**:=**" in **Tutorial D** (and some other languages). Example 2.7 shows a simple invocation of assignment to "update" the variable `SN`.

Example 2.7: A Simple Assignment

```
SN := SID ( 'S2' ) ;
```

If that assignment is given some time after the declaration shown in Example 2.5, then the effect will indeed be to change the value of the variable `SN`. It is changed from being the student identifier S1 to the student identifier S2. Here `SID ( 'S2' )` is the *source* for the assignment and `SN` is the *target*. The source does not have to be a literal, of course, as Example 2.8 demonstrates.

Example 2.8: Assignment Source Not a Literal

```
SN := SID ( LEFT ( THE_C (SN), 1 ) || '5' ) ;
```

Explanation 2.8:

- **`THE_C (SN)`** denotes the value of the `C` component of the current value of `SN` under the possrep (`SID`) declared for type `SID`. `LEFT ( THE_C (SN), 1 )` therefore denotes the leftmost character of that string, which will of course be the letter S.
- **`||`** is the concatenation operator for strings. `LEFT ( THE_C (SN), 1 ) || '5'` concatenates the letter S with the digit 5 to produce the string `'S5'`. In Example 2.7 this string is then operated on by the `SID` selector to yield the corresponding value of type `SID`, which is assigned to `SN`. Note that the term, `SN`, is used on both sides of the assignment. On the left it denotes the variable itself, the target. On the right, where it appears as part of the source, it denotes the current value of `SN`.

Example 2.9 is an invocation of a hypothetical update operator other than assignment—probably a user-defined operator. As with assignment, the invocation takes the form of a command.

Example 2.9: Invocation of a user-defined update operator

```
CALL SET_DIGITS ( SN , 23 ) ;
```

Explanation 2.9:

- `CALL` announces that an update operator invocation follows. The effect of that invocation is also the effect of the `CALL` statement.
- `SET_DIGITS ( SN , 23 )` , let us assume, has the same effect as
  `SN := SID ( SUBSTRING ( THE_C (SN), 0, 1 ) || '23';`
  (which is almost identical to Example 2.8). Note that `SN` in the invocation of `SET_DIGITS` is an argument substituted for a parameter that is defined for update, so here stands for the variable itself and not for its current value.

Tutorial D additionally allows certain special kinds of expression to appear as update targets. Such expressions are called *pseudovariables*—they are not real variables but they can be treated as if they were. A pseudovariable takes the form of an invocation of a read-only operator in which one of the operands is a reference to a variable (or pseudovariable) that is to be updated.

Example 2.10: Assignment to a pseudovariable

```
THE_C ( SN ) := 'S2' ;
```

Explanation 2.10:

- **`THE_C ( SN )`** denotes the `C` possrep component of the current value of the variable `SN`. However, because the invocation of `THE_C` appears in a target position here, that value is to be replaced by the value denoted by the expression on the right-hand side of the assignment.
- **`'S2'`** is a literal of type `CHAR`, the declared type of the `C` possrep component defined for type `SID`.
- The entire statement is equivalent to the regular assignment
  `SN := SID ( 'S2' ) ;`
  —not a very compelling example of pseudovariable updating, perhaps, but such shorthands are very useful when updating variables of more complicated types, having very many possrep components. Counterparts typically exist in object-oriented programming languages, in connection with object classes.

2.13 Conclusion

I conclude this chapter by reminding you of the important distinctions I drew to your attention at its beginning. I repeat them here, with illustrative examples to remind you:

- syntax and semantics (expressions and their meaning)
  An expression (syntax) *denotes* either a value or a variable.
- values and variables
  A value such as the integer 3, the character string `'London'`, or the relation `RELATION { TUPLE { A 3, B 'London' } }` is something that exists independently of time or space and is not subject to change. A variable *is* subject to change (in value only), by invocation of update operators. A variable reference, such as the expression `X`, denotes either the variable of that name or its current value, depending on the context in which it appears.
- values and representations of values
  The character string value denoted by `'S1'` is a *possible representation* of the student identifier S1, a value of type `SID` denoted by `SID('S1')`.
- types and representations
  `POSSREP { C CHAR }` defines a possible representation for all values of type `SID`.
- read-only operators and update operators
  "+" is a *read-only operator* because, when it is invoked, it returns a value. ":=" is an *update operator* because, when it is invoked, it has the effect of replacing the current value of a variable (i.e., *updating* the variable by *changing* its value)—and does *not* return a value.
- operators and invocations
  `SID` is an *operator.* Its signature is `SID(C CHAR)`. `SID('S1')` denotes an *invocation* of `SID`. Similarly, "+" is an operator, with signature `+(A RATIONAL, B RATIONAL)`, and `x+y` denotes an invocation of "+".
- parameters and arguments
  `C CHAR` is a *parameter* (and in fact the only parameter) of the operator `SID`. `'S1'` denotes the argument substituted for `C CHAR` in the invocation `SID('S1')`. Similarly, `x` and `y` denote the arguments substituted for `A RATIONAL` and `B RATIONAL`, respectively, in the invocation `x+y`.

You can now test your understanding of these distinctions by carrying out the first set of accompanying exercises (which also include some revision material for Chapter 1). In the second set you can start to explore **Tutorial D** in action, using *Rel.*

EXERCISES

Complete sentences 1–10 below, choosing your fillings from the following:

=, :=, ::=, argument, arguments, body, bodies, `BOOLEAN`, cardinality, `CHAR`, `CID`, degree, denoted, expressions, false, heading, headings, `INTEGER`, list, lists, literal, literals, operator, operators, parameter, parameters, read-only, set, sets, `SID`, true, type, types, update, variable, variables.

In 1–5, consider the expression `X = 1 OR Y = 2`.

1. In the given expression, = and `OR` are _____ whereas `X` and `Y` are _____ references.
2. `X` and `1` denote _____ to an invocation of _____.
3. The value _____ by the given expression is of _____ `BOOLEAN`.
4. `1` and `2` are both _____ of _____ `INTEGER`.
5. The operators used in the given expression are _____ operators.

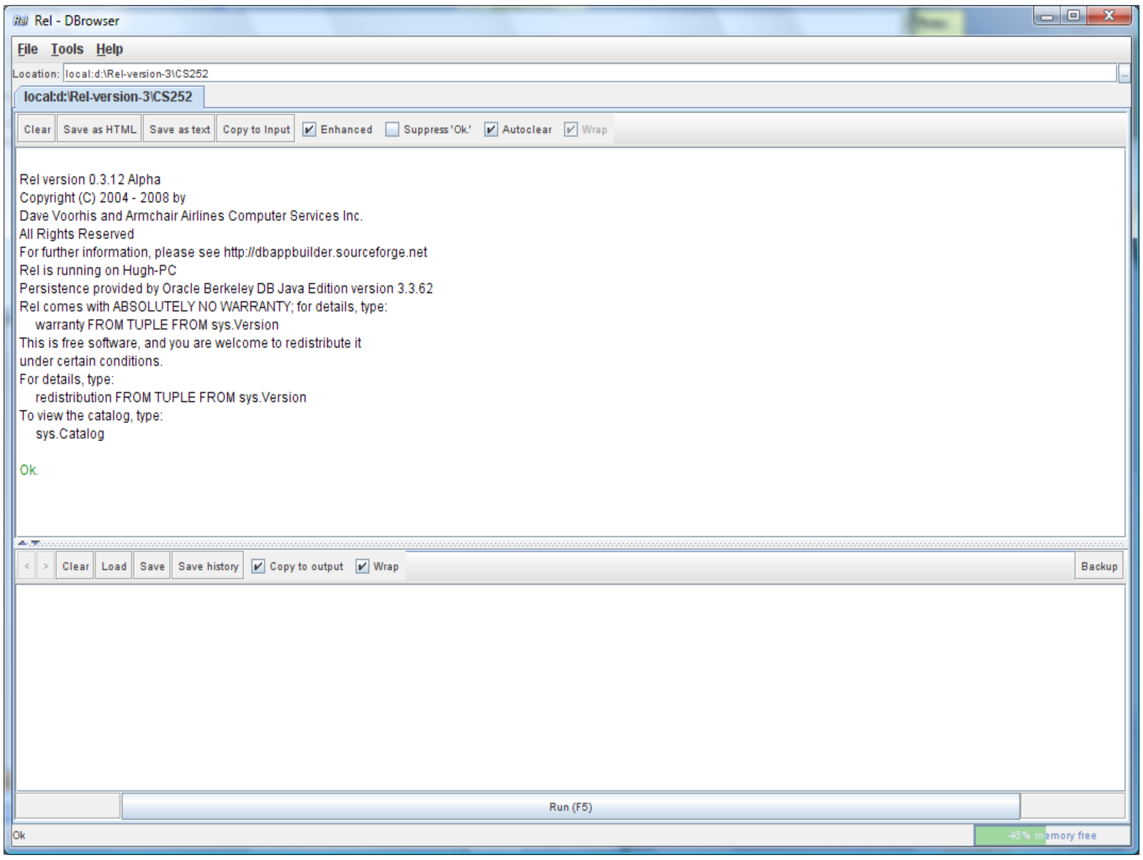
In 6–10, consider the expression `RELATION { X SID, Y CID } { }`.

6. It denotes a relation whose _____ is zero and whose _____ is two.
7. It is a relation _____.
8. The declared type of `Y` is _____.
9. In general, the heading of a relation is a possibly empty _____ of attributes and its body is a possibly empty _____ of tuples.
10. It is _____ that the assignment `RV __ RELATION { X SID, Y CID } { }` is legal if the _____ of `RV` is `{ Y CID, X SID }`, _____ that it is legal if the _______ of `RV` is `{ A SID, B CID }`, _____ that it is legal if the _____ of `RV` is `{ X CID, Y SID }`, and _____ that it is legal if the _____ of `RV` is `{ X CHAR, Y CHAR }`.

Getting Started with *Rel*

After you have downloaded and installed *Rel* from http://dbappbuilder.sourceforge.net/Rel.html, work through the following exercises. From number 7 onwards they involve constructs introduced in Chapter 4. You might prefer to wait until you have studied that chapter but on the other hand a little hands-on experience might help you to understand that chapter when you come to it.

1. Start up *Rel*'s DBrowser. DBrowser is the general-purpose client application provided by *Rel* for evaluating **Tutorial D** expressions and executing **Tutorial D** statements entered by the user.
2. Familiarise yourself with the way of working and the things you can do in *Rel*. You should be looking at a window something like this (which was obtained in Windows Vista):



```
Rel version 0.3.12 Alpha
Copyright (C) 2004 - 2008 by
Dave Voorhis and Armchair Airlines Computer Services Inc.
All Rights Reserved
For further information, please see http://dbappbuilder.sourceforge.net
Rel is running on Hugh-PC
Persistence provided by Oracle Berkeley DB Java Edition version 3.3.62
Rel comes with ABSOLUTELY NO WARRANTY; for details, type:
    warranty FROM TUPLE FROM sys.Version
This is free software, and you are welcome to redistribute it
under certain conditions.
For details, type:
    redistribution FROM TUPLE FROM sys.Version
To view the catalog, type:
    sys.Catalog

Ok.
```

- Note the layout of the window: a lower pane into which you can type statements to be executed, an upper pane in which results are displayed, and the movable horizontal bar between the panes.
- Note the ▲ and ▼ at the left-hand end of the horizontal bar, allowing you to let one or the other pane occupy the whole window for a while.
- See what is available on the Tools menu and perhaps choose your preferred font.
- Note the < and > to the left of the menu on the input (lower) pane. These are greyed out initially but after you have executed a couple of statements you will be able to use them to recall previously executed statements to the input pane.
- Note the toolbars on both panes. As you do the exercises, decide which options suit you best. Note that you can save the contents of either pane into a local file, and that you can load the contents of a local file into the input area.
- Note the check boxes on the right of the toolbars. They are fairly self-explanatory, apart from "Enhanced", which we will examine later.
- The box at the top of the upper pane, labelled "Location:", identifies the directory containing the database you are working with. You can switch to another directory by clicking on the little button to the right of the box, labelled with three dots (…).

3. Type the following into the lower pane:

```
output 2+2 ;
```

Execute what you have typed, either by clicking on Evaluate (F5) shown at the bottom of the window or by pressing F5.

Now delete the semicolon and try executing what remains. (If necessary, use the < button on the lower pane to recall the statement.) You will see how *Rel* handles errors.

Now strike out the word `output` and do not put back the semicolon. What happens when you execute that? (i.e., just `2+2`).

You have now learned:

- that in *Rel* (as in **Tutorial D**) every executable *command* (or statement) is terminated by a semicolon;
- that *Rel* allows you to obtain the result of evaluating an *expression* by using an `output` statement;
- that *Rel* treats an attempt to "execute" an expression `x` as shorthand for the statement `output x ;` — the absence of the semicolon signals to *Rel* that you are using this convenient shorthand.

4. This exercise is merely to alert you to a certain awkwardness in *Rel* that has no real importance but might cause you to waste a lot of time if you are not warned about it. It's the same as Step 3 except that instead of `2+2` you type `2+2.0`. Look closely at what happens. It doesn't work!

   Rel, like some other languages, treats `INTEGER` and `RATIONAL` as distinct types. If you want to do arithmetic on rational numbers, both operands must be rational numbers. Literals denoting rational numbers are distinguished from those denoting integers by the presence of a decimal point, and *Rel* follows the convention in the English-speaking community of using a full stop for this purpose (as opposed to the comma that is used throughout most of Europe, for example).

   Now try this: `1/2` (i.e., the integer 1 divided by the integer 2). And then this: `1.0/2.0`.

   You have now learned that (a) the operands of dyadic arithmetic operators in *Rel* must be of the same type, and (b) the type of the result of an invocation of such an operator is always of the same type as the operands. **Tutorial D** is silent on such issues, because they are orthogonal to what **Tutorial D** is really intended for (teaching relational theory). But every implementation of **Tutorial D** has to address them somehow.

   Fortunately, arithmetic is orthogonal to relational theory and there is no need for us to be bothered by *Rel*'s behaviour here. You have possibly already learned that the same problems do not arise in SQL, where `1/2`, `1/2.0` and `1.0/2.0` are all equivalent, in spite of the fact that `INTEGER` and `REAL` (SQL's counterpart of **Tutorial D**'s `RATIONAL`) are also distinct types in SQL.

5. Now try the following compound statement:
   ```
   begin ;
   VAR x integer init(0) ;
   x := x + 1 ;
   output x ;
   end ;
   ```

   Why do we have to write `output x ;` in full here, instead of just `x`?

   Now write the fourth line in uppercase: `OUTPUT X ;` What happens?

   Try `OUTPUT x ;` instead. What have you learned about *Rel*'s rules concerning *case sensitivity*?

6. Now you can start investigating *Rel*'s support for relations (though not relational databases yet). First, see how *Rel* displays a relation (i.e., the result of evaluating a relation expression) in its upper pane. *Rel* supports two styles of presentation, depending on whether the "Enhanced" option is checked.

   With "Enhanced" unchecked (it is usually checked to start with), get *Rel* to evaluate the following relation expression (a literal which we shall call `enrolment`):

   ```
   RELATION {
   TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne' },
   TUPLE { StudentId 'S1', CourseId 'C2', Name 'Anne' },
   TUPLE { StudentId 'S2', CourseId 'C1', Name 'Boris' },
   TUPLE { StudentId 'S3', CourseId 'C3', Name 'Cindy' },
   TUPLE { StudentId 'S4', CourseId 'C1', Name 'Devinder' }
   }
   ```

   See Section 2.9. Look closely at the output. Is it identical to the input?

   Next, without altering the contents of the lower pane, turn "Enhanced" back on. Note the effect on the display in the output pane.

   Now delete all the tuple expressions, leaving just `RELATION { }`. What happens when *Rel* tries to evaluate that?

   Now use < to recall the original `RELATION` expression to the input pane and re-evaluate it with "Enhanced" *off*. Use copy-and-paste to copy the result to the input pane, then delete all the `TUPLE` expressions, to leave this:

   ```
   RELATION {StudentId CHARACTER, CourseId CHARACTER,
             Name CHARACTER} { }
   ```

   Study the result of that in the output pane, first with "Enhanced" off, then with it on.

   What conclusions do you draw from all this, about *Rel* and **Tutorial D**?

   From now on you can run with "Enhanced" either on or off, according to your own preference.

   Next, enter the following literal, perhaps by using the < button to recall `enrolment` and editing it:

```
RELATION {
TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne' },
TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne' }
}
```

Before you press Evaluate (F5), think about what you expect to happen. Does the result meet your expectation? How do you explain it?

Use < again to recall the `enrolment` literal. Insert the word `WITH` at the beginning, add `AS enrolment : enrolment` at the end, to give:

```
WITH ( enrolment :=
RELATION {
  TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne' },
  TUPLE { StudentId 'S1', CourseId 'C2', Name 'Anne' },
  TUPLE { StudentId 'S2', CourseId 'C1', Name 'Boris' },
  TUPLE { StudentId 'S3', CourseId 'C3', Name 'Cindy' },
  TUPLE { StudentId 'S4', CourseId 'C1', Name 'Devinder' }
} ) : enrolment
```

and evaluate that.

How do you understand what you have just done? (`WITH` isn't described in the book. In case you aren't clear, try this in *Rel*: `WITH ( four := 2+2, eight := four+four ) : eight + four`. Note carefully that the introduced names, four and eight, are local only.)

By inspection of `enrolment` only, write down all the cases you can find of two students such that there is at least one course they are both enrolled on.

7. For this exercise you will need to continue using < to recall your previous command (now including the definition of the introduced name `enrolment`) and overtype as necessary. Use `enrolment` to investigate the relational operator known as **projection (*see Chapter 4, Section 4.6*).** The projection of a given relation over a specified subset of its attributes yields another relation. In **Tutorial D** a projection is specified by writing a list of attribute names, enclosed in braces { } and separated by commas, after the operand relation. The key words `ALL BUT` can optionally precede the list of attribute names, inside the braces.

   How many distinct projections can be obtained from `enrolment`? Obtain as many of these as you wish, trying both the "inclusion" method and the "exclusion" method using `ALL BUT`.

8. Still using `enrolment`, investigate the relational operator known as **rename (*see Chapter 4, Section 4.5*).** The renaming of a given relation returns that relation with one or more of its attributes renamed. In **Tutorial D** a renaming is specified by writing `RENAME {` *`old`* `AS` *`new`*`, ... }` after the operand relation.

   At the moment you should have this in your input pane:

```
WITH ( enrolment :=
RELATION {
  TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne' },
  TUPLE { StudentId 'S1', CourseId 'C2', Name 'Anne' },
  TUPLE { StudentId 'S2', CourseId 'C1', Name 'Boris' },
  TUPLE { StudentId 'S3', CourseId 'C3', Name 'Cindy' },
  TUPLE { StudentId 'S4', CourseId 'C1', Name 'Devinder' }
} ) : enrolment
```

Replace the single word (`enrolment`) that follows the colon by a renaming of `enrolment` such that the result has attribute name `SID1` instead of `StudentId`, `N1` instead of `Name`, and is otherwise the same as `enrolment` itself. Replace the : that ends the `WITH` specification by `E1` := and add : `E1` at the end. The result should look like this:

```
WITH ( enrolment :=
RELATION {
  TUPLE { StudentId 'S1', CourseId 'C1', Name 'Anne' },
  TUPLE { StudentId 'S1', CourseId 'C2', Name 'Anne' },
  TUPLE { StudentId 'S2', CourseId 'C1', Name 'Boris' },
  TUPLE { StudentId 'S3', CourseId 'C3', Name 'Cindy' },
  TUPLE { StudentId 'S4', CourseId 'C1', Name 'Devinder' }
} , E1 := <your renaming of enrolment, as specified> ) :
E1
```

Evaluate that to check that you wrote the renaming correctly.

9. Now replace the : by , `E1` := and this time add a similar renaming of `enrolment`, using `SID2` and `N2` instead of `SID1` and `N1` for the new attribute names, and add : `E1 JOIN E2` at the end. You are investigating the operator called `JOIN` ***(see Chapter 4, Section 4.4)***.

   How do you interpret the result? How many tuples does it contain? Replace the key word `JOIN` by `COMPOSE` (***see Chapter 5, Section 5.2***). How do you interpret *this* result? How many tuples are there now? How do you account for the difference?

10. Add `WHERE NOT ( SID1 = SID2 )` to end of the expression you evaluated in Step 9 (***see Chapter 4, Section 4.7***). Examine the result closely. Now place parentheses around `E1 COMPOSE E2` and evaluate again. Confirm that you get the same result.

    Repeat the experiment, replacing `WHERE NOT ( SID1 = SID2 )` by `{ SID1 }`. Do you get the same results this time? If not, why not?

    What does all this tell you about operator precedence rules in **Tutorial D**?

    Why was it probably a good idea to add that `WHERE` invocation? Did it completely solve the problem? If not, can you think of a better solution?

    What connection, if any, do you see between this exercise and Exercise 6?

11. Load the file `OperatorsChar.d`, provided in the Scripts subdirectory of the *Rel* program directory, and execute it. Now you have the operators used in Example 2.4, among others. Give appropriate type definitions for types `NAME` and `CID`. Notice that the operator `TO_UPPER_CASE` is available for converting a given string to its upper-case counterpart. You might like to try using this operator to define a constraint for type `NAME` to ensure that all names begin with a capital letter.

12. Close *Rel* by clicking on File/Exit.

3 Predicates and Propositions

3.1 Introduction

In Chapter 1 I defined a database to be "…an *organised*, machine-readable collection of *symbols*, to be *interpreted* as a *true* account of some *enterprise*." I also gave this example (extracted from Figure 1.1):

StudentId	Name	CourseId
S1	Anne	C1

I suggested that those green symbols, organised as they are with respect to the blue ones, might be understood to mean:

> "Student S1, named Anne, is enrolled on course C1."

In this chapter I explain exactly how such an interpretation can be justified. In fact, I describe the general method under which data organized in the form of relations is to be interpreted—to yield *information*, as some people say. This method of interpretation is firmly based in the science of *logic*. Relational database theory is based very directly on logic. Predicates and propositions are the fundamental concepts that logic deals with.

Fortunately, we need to understand only the few basic principles on which logic is founded. You may well already have a good grasp of the principles in question, but even if you do, please do not skip this chapter. For one thing, the textbooks on logic do not all use exactly the same terminology and I have chosen the terms and definitions that seem most suitable for the purpose at hand. For another, I do of course concentrate on the points that are particularly relevant to relational theory; you need to know which points those are and to understand exactly why they are so relevant.

3.2 What Is a Predicate?

Predicates, one might say, are what logic is all about. And yet the textbooks do not speak with one voice when it comes to pinning down *exactly* what the term refers to! I choose the definition that appears to me to fit best, so to speak, with relational database theory. We start by looking again at that possible interpretation of the symbols S1, Anne, and C1, placed the way they are in Figure 1.1:

> "Student S1, named Anne, is enrolled on course C1."

This is a *sentence*. Sentences are what human beings typically use to communicate with each other, using language. We express our interpretations of the data using sentences in human language and we use relations to organize the data to be interpreted. Logic bridges the gap between relations and sentences.

Our example sentence can be recast into two simpler sentences, “Student S1 is named Anne” and “Student S1 is enrolled on course C1”. Let’s focus on the second:

Example 3.1: A simple sentence

“Student S1 is enrolled on course C1.”

The symbols S1 and C1 appear both in this sentence and in the data whose meaning it expresses. Because they each designate, or refer to, a particular thing—S1 a particular student, C1 a particular course—they are called *designators*. The word Anne is another designator, referring to a particular forename. “An Introduction to Relational Database Theory” is also a designator, referring to a particular book, and so is, for example, -7, referring to a particular number.

Now, suppose we replace S1 and C1 in Example 3.1 by another pair of symbols, taken from the same columns of Figure 1.1 but a different row. Then we might obtain

Example 3.2:

“Student S3 is enrolled on course C3.”

A pattern is clearly emerging. For every row in Figure 1.1, considering just the columns headed StudentId and CourseId, we can obtain a sentence in the form of Examples 3.1 and 3.2. The words “Student…is enrolled on course…” appear in that order in each case and in each case the gaps indicated by…—sometimes called placeholders—are replaced by appropriate designators. If we now replace each placeholder by the name given in the heading of the column from which the appropriate designator is to be drawn, we obtain this:

Example 3.3:

“Student *StudentId* is enrolled on course *CourseId*.”

Example 3.3 succinctly expresses the way in which the named columns in each row of Figure 1.1 are probably to be interpreted. And we now know that those names, StudentId and CourseId, in the column headings are the names of two of the attributes of the relation that Figure 1.1 depicts in tabular form.

Now, the sentences in Examples 3.1 and 3.2 are in fact *statements*. They state something of which it can be said, “That is true”, or “That is not true”, or “I believe that”, or “I don’t believe that”.

Not all sentences are statements. A good informal test, in English, to determine whether a sentence is a statement is to place "Is it true that" in front of it. If the result is a grammatical English *question*, then the original sentence is indeed a statement; otherwise it is not. Here are some sentences that are not statements:

- "Let's all get drunk."
- "Will you marry me?"
- "Please pass me the salt."
- "If music be the food of love, play on."

They each fail the test. In fact one of them is a question itself and the other three are imperatives, but we have no need of such sentences in our interpretation of relations because we seek only information, in the form of statements—statements that we are prepared to believe are statements of fact; in other words, statements we believe to be *true*. We do not expect a database to be interpreted as asking questions or giving orders. We expect it to be stating facts (or at least what are believed to be facts).

As an aside, I must own up to the fact that some sentences that would be accepted as statements in English don't really pass the test as they stand. Here are two cases in point, from Shakespeare:

- "O for a muse of fire that would ascend the highest heaven of invention."
- "To be or not to be—that is the question."

Hva får egentlig en ingeniør- eller teknologistudent for 300 kroner?
• Medlemskap i en aktiv studentorganisasjon – hele studietiden
• 150 tillitsvalgte studenter som ivaretar dine interesser
• Jobbsøkerkurs
• Gratis PC-forsikring og gode bank- og forsikringstilbud
• Teknisk Ukeblad og NITO Refleks
• Møteplasser på web 2.0
Flere medlemsfordeler og innmelding: www.nito.no/student
Alle som studerer på ingeniør-, bioingeniør-, sivilingeniør eller andre teknologistudier (høgskolekandidat, bachelor eller master) kan bli medlem i NITO.
NITO NORGES STØRSTE ORGANISASJON FOR INGENIØRER OG TEKNOLOGER

The first appears to lack a verb, but we know that "O for a…" is a poetical way of expressing a wish for something on the part of the speaker, so we can paraphrase it fairly accurately by replacing "O" by "I wish", and the sentence thus revised passes the test. In the second case we have only to delete the word "that", whose presence serves only for emphasis (and scansion, of course!), and alter the punctuation slightly: "It is true that 'to be or not to be?' is the question."

Now, a statement is a sentence that is *declarative* in form: it declares something that is supposed to be true. Example 3.3, "Student *StudentId* is enrolled on course *CourseId*", is not a statement—it does not pass the test. It does, however, have the grammatical form of a statement. We can say that, like a statement, it is declarative in form. And we know that we have only to replace those italicised symbols (about which more anon) by appropriate designators, such as S1 and C1, to make it into a statement. Now I can say exactly what are meant by the terms *predicate* and *proposition*, starting with *predicate*.

A sentence that is in declarative form has a certain meaning, hopefully agreed upon by all who might read or hear it. That meaning is what logicians call a predicate. We can therefore say that such a sentence *denotes* a certain predicate. It is important to bear the distinction between the sentences and the predicates they denote firmly in mind. For consider the following sentences:

- 1 is less than 2
- 1 est moins que 2
- 1 < 2

They are written in three different languages but they all have exactly the same meaning—they denote the same predicate. Here are some more declarative sentences:

- I love you.
  Here the designators are the pronouns, "I" and "you". In isolation they designate nothing but in an appropriate context they do, if we know who the speaker is and to whom the sentence is spoken.
- The present king of France is bald
  This is a popular example used by logicians when they want to discuss problems concerning designators. Here we have something—"The present king of France"—that looks like a designator but in fact, at the time of writing, designates nothing because France has no king. At various times in the past, however, France *has* had a king. As far as relational databases are concerned, problems to do with designation are addressed by types (see Chapter 2) and database constraints (Chapter 5).
- 2 + 2 = 5
- $x < y$
- $a + b = c$
  These three are sentences written in mathematical notation. The first is a statement but the other two are not—they contain italicised symbols that designate nothing.

- Student *s* is enrolled on course *c*.
  This is identical to Example 3.3, except that *s* replaces *StudentId* and *c* replaces *CourseId*. It will suit our present purposes to regard this and Example 3.3 as denoting distinct predicates, though not all textbooks on logic take this stance (and not all are even clear on the matter).
- *P(x,y)*
  This kind of notation is commonly used by logicians as denoting a predicate without stating which particular predicate is being denoted. Notice that the symbol *P*, standing for what would be written in roman for a particular predicate, is italicised. For example, if we replace *P* by the "less than" symbol, <, we obtain *<(x,y),* which might be just another way of denoting the same predicate as $x < y$.

Up to now I have been very careful to maintain that clear distinction between the sentences and the predicates denoted by those sentences. However, it is often very convenient to refer to the sentences themselves as predicates, just to avoid excessive repetition of "denoted by", and I will do so frequently from now on in this book—starting right now.

Consider again, then, the predicate "Student *StudentId* is enrolled on course *CourseId*" (Example 3.3). Instead of designators for student and course it has symbols *StudentId* and *CourseId*, neither of which designates anything in particular. They are usually called *variables*, but note very carefully that they are not variables in the sense of that term as defined in Chapter 2. Logic does not deal with that kind of variable, so no confusion arises in texts dealing with logic alone, but in this book we have to deal with both kinds of variable. For that reason I prefer the alternative term, *parameter,* for variables appearing in predicates—but we shall see later that although a parameter is a variable, not all variables appearing in predicates are parameters (so I will occasionally have to revert to the term variable).

Special adjectives are used to indicate the number of parameters in a predicate. In general these take the form of a number suffixed by "-adic". Thus a 5-adic predicate has five parameters, a 0-adic predicate none at all, and an *n*-adic predicate has *n* parameters. For the lower numbers the appropriate prefix derived from Greek is often used instead: monadic, dyadic, triadic, tetradic, and so on, though we switch to Latin with niladic for 0-adic.

Some of the predicates I have shown you contain one or more parameters; others do not. Those that contain no parameters are, as already noted, statements. The predicate denoted by a statement is a very important special case: what logicians call a *proposition*. Now, recall that test used to determine whether a sentence in English is in fact a statement—can it be prefixed by "Is it true that" to yield a grammatical question in English? A proposition, then, being denoted by a sentence *p* that passes that test, is something that is either true or false, depending on the correct answer to the question, "Is it true that *p*?"

3.3 Substitution and Instantiation

If a predicate has *n* parameters (*n*>0) and we replace one of those parameters by a designator, we obtain a predicate with *n*-1 parameters. For example, in the dyadic predicate

$a < b$

if we replace *b* by 10 we obtain the monadic predicate

$a < 10$

We say that the designator 10 is *substituted* for the parameter *b*. If the designator is being substituted for a parameter that appears more than once in the sentence denoting the predicate, then of course that same designator must be substituted for each such appearance. For example, in the triadic predicate

$a < b$ and $b < c$

if we substitute 10 for *b* we obtain the dyadic predicate

$a < 10$ and $10 < c$

If we substitute designators for all of the parameters, we obtain a niladic predicate—a proposition. For example, if we substitute 5 for *a* and 15 for *c* in the dyadic predicate above, we obtain

5 < 10 and 10 < 15

which happens to denote a true proposition. The act of substituting designators for all the parameters of a predicate is sometimes referred to as *instantiation*—a term that is especially useful in the interpretation of relations. We can say, then, that the proposition 5 < 10 is an instantiation of the predicate $a < 10$. It is also an instantiation of the predicates $5 < b$, $a < b$ and $x < y$. (Some textbooks use the term *full instantiation*, regarding substitution of some but not all of the parameters as "partial" instantiation. Personally, I find the notion of a "partial instance" (of a kind) rather weird!)

According to our stance of regarding a proposition as a special case of a predicate, we must also accept the strange-looking notion that 5 < 10 is an instantiation—the only possible instantiation—of itself! Such quirky observations are not often found in textbooks on logic, but they often turn out to be more important than might appear at first sight when it comes to designing computer languages, as we shall eventually see.

Extension and Intension

Now, given some predicate *p*, we can consider all the possible instantiations of *p*. Each is either a true proposition or a false one. The true instantiations of *p*, taken collectively, are referred to as the *extension* of *p*. This concept will prove to be very important to us in the database context. Also important is the *intension* of a predicate. Loosely speaking this is just its meaning, but note that predicates that differ only in the names of their parameters—for example, $a < b$ and $x < y$, have the same intension, and this gives meaning in turn to their instantiations. The same term, intension, is also used for the meaning of a designator. Thus, although 5 < 10 and 10 < 5 are both instantiations of $a < b$, their meanings are obviously different (as are their truth values). Intension is important with regard to our interpretation of a relation, but the relation itself tells us nothing about what its tuples might mean.

3.4 How a Relation Represents an Extension

In Chapters 1 and 2 the word "set" appears many times, without definition but intended to be precise. For example, the heading of a relation is described as a set of attributes, its body a set of tuples. I assume you are somewhat familiar with the concept of a set and in any case this book does not include a complete account of the mathematical theory of sets. The theory of sets arose out of predicate logic (in the 19th century); the theory of relations arose out of the theory of sets; and the theory of relational databases arose out of the theory of relations. The remainder of this section explains these connections.

A set is a collection of distinct objects, termed its *elements.* Each element appears exactly once—there is no sense in which the same element can appear more than once in a given set. Anything can be an element of some set. Even a set can be an element of a set, though we can run into trouble, as the philosopher Bertrand Russell famously observed, if we consider the possibility of a set being one of its own elements.

Mathematicians recognize two distinct methods of defining (or denoting) a set. These methods relate to the terms extension and intension that we have just met, and are indeed called *extensional definition* and *intensional definition.* We are interested in both methods.

An extensional definition simply enumerates the elements. In mathematical notation the elements are denoted by a list of designators enclosed in braces. The order in which the designators are written is insignificant. The same element can be designated more than once in the list but the "extra" designations signify nothing. Thus, the following extensional definitions all denote the same set:

Example 3.4: extensional definitions of a set

{ 2, 3, 5 }
{ 5, 2, 3 }
{ 2, 5, 3, 2, 2, 5 }

You have seen several extensional definitions already, in Chapter 2. For example, in **Tutorial D** names for relation types, such as `RELATION { StudentId SID, CourseId CID }` and `RELATION { a INTEGER, b INTEGER, c INTEGER }`, the word `RELATION` is followed by an extensional definition for a heading, in which each element is an attribute, designated by an attribute name paired with a type name.

The empty set is often denoted in mathematical texts by the symbol Φ (one of several graphemes for the Greek letter phi) but **Tutorial D** uses its extensional definition, `{ }`, as in the type name `RELATION { }` mentioned in Chapter 2.

Tutorial D also allows the body of a relation to be denoted by an extensional definition, as in Example 2.3 in Chapter 2, repeated here for convenience:

```
RELATION {
  TUPLE { StudentId SID('S1'), CourseId CID('C1'),
          Name NAME('Anne')},
  TUPLE { StudentId SID('S1'), CourseId CID('C2'),
          Name NAME('Anne')},
  TUPLE { StudentId SID('S2'), CourseId CID('C1'),
          Name NAME('Boris')},
TUPLE { StudentId SID('S3'), CourseId CID('C3'),
        Name NAME('Cindy')},
TUPLE { StudentId SID('S4'), CourseId CID('C1'),
        Name NAME('Devinder')}
}
```

Each element of the body is denoted by a tuple literal consisting of the word `TUPLE` followed by an extensional definition for the tuple itself. Each element of the tuple is denoted by an attribute name paired with a literal denoting some value of the declared type of that attribute.

An intensional definition for a set specifies a rule or property. The set being defined consists of every object that obeys the specified rule or has the specified property. For example, we have the set of all dogs, whose elements are all and only those objects that have the property of being a dog, and the set of all numbers from 1 to 10, whose elements are precisely those numbers *n* that obey the rule "*n* is between 1 and 10, inclusive". Of course we could equally well say that the set of all dogs is the set whose elements are all and only those objects that obey the rule "*x* is a dog", and the set of all numbers from 1 to 10 is the set whose elements are precisely those numbers that have the property of being between 1 and 10, inclusive.

It should come as no surprise, now, to learn that the rule or property is actually some expression denoting a predicate. In fact, the mathematical notation for writing an intensional definition, sometimes called *set-builder notation*, consists of an expression specifying the parameters of a predicate and the predicate itself. Example 3.5 gives some intensional definitions using this notation.

Example 3.5: intensional set definitions

{ x : x is a dog }
{ n : n is an integer and $1 \leqslant n \leqslant 10$ }
{ p : p is prime and $p<7$ }
{ x : $0<x<9$ and $x = 1 + 2^n$ for some integer n, where $n \geq 0$
{ q : q is prime and $q<7$ }

In set-builder notation, the braces indicate that a set is being defined. The colon can be pronounced "such that". The elements of the set being defined, also known as its members, are all and only those objects that *satisfy* the given predicate, sometimes referred to as a *membership predicate* for that set. The last three definitions in Example 3.5 all define the same set, that being the set defined in Example 3.4.

Each of the definitions in Example 3.5 uses a monadic predicate, with parameter name *x*, *n*, *p*, *x* again, and *q*, respectively. (The *n* appearing in the penultimate example is not a parameter, as will be explained shortly.) The sets denoted by these examples are not relations, nor are they bodies of relations. The elements of { 2, 3, 5 }, for example, are all numbers, and the body of a relation consists of tuples, not numbers. We are very close to pinning down a method of interpreting a relation but we haven't quite got there yet. The light should dawn when we look at set-builder notation using predicates with more than one parameter, such as those in Example 3.6.

Example 3.6: more intensional definitions

{ $\langle a, b \rangle : a < b$ }
{ $\langle x, y, z \rangle : x + y = z$ }
{ <*StudentId, CourseId*> : Student *StudentId* is enrolled on course *CourseId* }

The example { $\langle a, b \rangle : a < b$ } can be read as "the set consisting of pairs of *a* and *b* such that *a* is less than *b*". A pair is a tuple: a 2-tuple, to be precise. The body of a relation is a set of tuples. The tuples of the body of a relation all have the same heading. The expression <*a, b*> can be thought of as specifying the names of the attributes of a heading. The objects <*a, b*> that satisfy $a < b$ are all tuples consisting of an *a* value and a *b* value.

We need to be clear, now, about that important term *satisfies*:

- An object *x* satisfies monadic predicate *P* if and only if substitution of *x* for the sole parameter of *P*, thus instantiating *P*, yields a true proposition.
- An n-tuple *t* satisfies *n*-adic predicate *P* if and only if substituting each element of *t* for the corresponding parameter of *P*, thus instantiating *P*, yields a true proposition

And now we need to be clear about that correspondence between tuple elements and parameters.

In mathematics, notation such as <1, 2>, using angle brackets, is often used to denote a tuple. Using this to instantiate the predicate $a < b$ relies on an ordering of the parameters *a* and *b*, without which we would not know whether <1, 2> is to be interpreted as 1<2 or, instead, 2<1. In relational databases we sometimes deal with relations of quite high degree, when reliance on some specific ordering would give rise to difficulty that is avoided by the use of attribute names. Thus, in **Tutorial D**, the tuple literals `TUPLE {a 1, b 2}` and `TUPLE {b 2, a 1}` both denote the tuple <1, 2> in the present context (and might instead denote <2, 1> in some other context). Whereas <1, 2> can represent an instantiation of any dyadic predicate, `TUPLE {a 1, b 2}` can represent an instantiation only of a predicate whose parameters are named *a* and *b*.

Now, with monadic predicates the problem of matching designators to the right parameters obviously doesn't arise, there being only one parameter to choose from. When considering whether the object named Rover satisfies the predicate "*x* is a dog", there is only the parameter *x* for which Rover can be substituted. Although the intensional definition *could* be written as { <*x*> : *x* is a dog } instead of as shown in Example 3.5, the mathematician might have little or no cause to include those angle brackets. However, as we shall see in the next chapter, uniformity is of the essence in a computer language, and a computer language treating relations of degree 1 differently from the others would be needlessly complicated and inconvenient to use. Thus, we revise the intensional definitions given in example 3.5 as shown in Example 3.7.

Example 3.7: revision of Example 3.5

{ <*x*> : *x* is a dog }
{ <*n*> : *n* is an integer and $1 \leqslant n \leqslant 10$ }
{ <*p*> : *p* is prime and $p<7$ }
{ <*x*> : $0<x<9$ and $x = 1 + 2^n$ for some integer *n* }
{ <*q*> : *q* is prime and $q<7$ }

The set-builder expressions in Example 3.7 all define sets of tuples—1-tuples, to be precise. Now we can't be so sure that the last three all denote the same set. The mathematician might say that they do, the set being {<2>, <3>, <5> } in each case, where the elements are "ordered" 1-tuples. But in relational database theory they do not: the tuples `TUPLE { p 2 }`, `TUPLE { x 2 }` and `TUPLE { q 2 }` are three different tuples.

We can usefully extend the definitions of *satisfies* (of tuples and predicates) to cover cases where the tuple has more elements than the predicate has parameters:

> A tuple *t* satisfies *n*-adic predicate *P* if and only if instantiating *P* by substituting a corresponding element of *t* for each of its parameters yields a true proposition.

For example, we can now say that `TUPLE { a 1, b 2, c 3 }` satisfies the predicate $a < b$. It also satisfies $c > b$, $b < 4$, and, for that matter, the niladic predicate $4 < 5$ (but not $5 < 4$—why not?). Note that under this definition a tuple that satisfies *P* must have at least as many elements as *P* has parameters.

Confining our attention to tuples as the objects that might or might not satisfy predicates entails absolutely no loss of generality. In fact, it might even be considered to gain something, for consider Example 3.8.

Example 3.8: intensional definitions using niladic predicates

{ < > : 2 < 1 }
{ < > : Student S1 is enrolled on course C1 }

The first defines the empty set, { }, because there is no 0-tuple that satisfies the given predicate. (There is only one 0-tuple, < >. It is denoted by the literal `TUPLE` { } in **Tutorial D**.) Assuming that student S1 is indeed enrolled on course C1, the second denotes the singleton set { < > }, this being the body of the relation `RELATION { TUPLE { } }`—more about this strange-looking relation in Chapter 4. The mathematician has little use for it, to be sure, but computer languages that fail to recognize special limiting cases tend to give rise to occasional but needless disappointment.

Now, at last, we can answer the question implied by the title of this section. Quite simply:

A relation represents the extension of some *n*-adic predicate *P* by having a body consisting of every *n*-tuple that satisfies *P*.

Finally, and most importantly, note that the extension of a predicate can vary from time to time. Indeed, relational databases are expected to consist of variables—relvars, in fact—whose assigned values do change from time to time; and yet the predicate whose extension is represented by the value assigned to such a relvar remains constant over time. Thus, we can and do speak of *the* predicate for a relvar (and note in passing that it might be quite useful for such predicates to be recorded in the system catalogue).

3.5 Deriving Predicates from Predicates

We have already seen one method by which a predicate can be derived from a given predicate, namely, substitution. And we have begun to see the relevance of substitution in the context of relational databases. There are two other general methods, both also relevant in this context. One is by use of the *logical connectives* of the propositional calculus, such as those denoted in many computer languages, including **Tutorial D**, by `AND`, `OR`, and `NOT`. The other is by *quantification*.

Why are ways of deriving predicates from predicates important to us? Well, the next chapter describes a set of operators for deriving relations from relations. This set of operators—the relational algebra—is arguably the most important component of the machinery we use to work with a relational database. If relations *r1* and *r2* represent predicates *p1* and *p2* and we can derive relation *r3* from *r1* and *r2*, then *r3* represents some predicate *p3* that can be derived from *p1* and *p2.* Moreover, the definitions of the relational operators we use to derive *r3*, as we shall see in Chapter 4, tell us how to derive *p3.* Therefore, if we know *p1* and *p2*, which tell us how to interpret *r1* and *r2*, we will know how to interpret *r3.* Conversely, if we can express *p3* in terms of *p1* and *p2,* then we can work out how to express *r3* in terms of *r1* and *r2.*

Logical Connectives

In human languages such as English we can connect two sentences together such that the result is itself a sentence. The connecting word is called a conjunction. For example, in "It's raining and I'm wet through", the conjunction "and" connects the sentences "It's raining" and "I'm wet through". Other examples:

> It's raining but I have my umbrella with me. ("but")
> I have my umbrella with me because it's raining. ("because")
> I shall have my umbrella with me if it rains. ("if")
> I shan't have my umbrella with me unless it rains. ("unless")

Since we use sentences to denote predicates it is easy to see that a sentence formed from two sentences in this way can denote a predicate formed from two predicates. And the meaning of the predicate thus formed can be derived from the meaning of the two constituent predicates and the meaning of the conjunction that connects the sentences.

Many of the so-called logical connectives of the propositional calculus correspond approximately to conjunctions in human languages such as English. `AND` and `OR` are obvious examples. But some of them—for example, `NOT`—just modify a single predicate and thus don't really do any connecting as such. To add to the possible confusion, the abstract noun used to refer to one of them in particular—`AND`—is called *conjunction*. The correspondence between logical connectives and conjunctions in human language is approximate because the meaning of a logical connective is always the same, regardless of the context in which it is used. Consider, for example, the English sentence, "You pay me by Tuesday or I'll sue." If that "or" were the logical connective, then the sentence is true when either you do pay me by Tuesday or I sue you; so you could pay me on Monday and I might still sue you. But this particular use of "or" would normally be taken to mean that either you pay me by Tuesday and I don't sue you, or you do not pay me by Tuesday and I do sue you—the so-called "exclusive or".

Negation (NOT)

In English, given a statement, s, we can derive the statement "It is not true that s." This denotes the negation of the proposition p denoted by s. Negation is usually denoted in formal treatments by the symbol $\neg$: the negation of p is written as $\neg p$. If p is true, then its negation is false; and if p is false, then its negation is true. It follows that $\neg\neg p$ has the same truth value as p.

The definition of a logical connective is usually presented as a *truth table*, in which the letters `T` and `F` stand for `TRUE` and `FALSE`, respectively. The truth table for negation is shown in Figure 3.1.

p	**¬*p***
T	F
F	T

Figure 3.1: The Truth Table for Negation

As we have seen, a proposition is a special case of a predicate, namely, a predicate with no parameters (a niladic predicate). Negation can be applied to other predicates too, the result always being a predicate with the same parameters as the predicate to which it is applied. Of course, a predicate with parameters has no truth value; therefore neither does its negation. However, the truth table for negation tells us that if tuple *t* satisfies predicate *p*, then *t* does *not* satisfy ¬*p;* and, conversely, if *t* fails to satisfy *p*, then *t* satisfies ¬*p*.

Consider the negation, "Student *StudentId* is **not** enrolled on course *CourseId*." Assuming the truth expressed by the `ENROLMENT` relation depicted in Figure 1.2 of Chapter 1, we can see that the tuple `TUPLE { StudentId SID('S1'), CourseId CID('C1') }` fails to satisfy this predicate. Here, by contrast, as some that do satisfy it:

> `TUPLE { StudentId SID('S1'), CourseId CID('C3') }`
> (No tuple in `ENROLMENT` shows student S1 as being enrolled on course C3)
>
> `TUPLE { StudentId SID('S2'), CourseId CID('C97') }`
> (No tuple in `ENROLMENT` shows S1 as being enrolled on C97 and in fact there is currently no such course, though C97 is a valid course identifier)
>
> `TUPLE { StudentId SID('S98'), CourseId CID('C97') }`
> (No tuple in ENROLMENT shows S98 as being enrolled on C97 and in fact there is currently no such student, though S98 is a valid student identifier)

In **Tutorial D**, as previously noted, negation is denoted by the key word `NOT`. For example, the expression `NOT (x = 5)` evaluates to `TRUE` for all values of `x` except 5, when it evaluates to `FALSE`.

You will have realised by now that the cardinality of a relation representing the extension of "Student *StudentId* is not enrolled on course *CourseId*" is likely to be very high indeed—almost certainly too high to be manageable using a computer. In Chapter 4 you will see that a relational DBMS's support for relations representing negated predicates is subject to a certain restriction, addressing this problem. But you may also be thinking that relations for negated predicates such as the example at hand would be of little use in practice, in which case you will agree, when you see the restriction I am referring to, that it is of little or no import.

Conjunction (AND)

In English, two sentences may be connected by the conjunction, "and", yielding a single sentence. For example, "Come in and make yourself at home", where the conjunction connects two imperatives, meaning that the person being spoken to is being enjoined to do both of those things. When the sentences being connected are statements, denoting propositions, the result is a single statement, denoting a single proposition. For example: "It's raining and I'm wet through". If it really is raining, and I really am wet through, then that is a true statement; otherwise—either it's not raining, or I'm not wet through—it is a false statement. That example illustrates the logical connective `AND`, usually denoted in formal treatments by the symbol $\wedge$. The connection of two propositions in this particular way is called *conjunction.* Unfortunately, as previously noted, there are other conjunctions in English whose use does not denote logical conjunction (for example, "or"). There are also other conjunctions that *do* denote logical conjunction (for example, "but").

The conjunction of propositions *p* and *q* is true if and only if *p* is true and *q* is true, as shown in Figure 3.2, the truth table for conjunction.

p	***q***	***p∧q***
T	T	T
T	F	F
F	T	F
F	F	F

Figure 3.2: The Truth Table for AND

As we have seen, a proposition is a special case of a predicate, namely, a predicate with no parameters (a niladic predicate). Conjunction can be applied to other predicates too, the result being a predicate. Again, a predicate with parameters has no truth value; therefore neither does its conjunction with some other predicate. However, the truth table for conjunction tells us that if tuple *t1* satisfies predicate *p1* and tuple *t2* satisfies predicate *p2*, then tuple *t3,* consisting of every element that is an element of either *t1* or *t2* (or both) satisfies *p1∧p2;* conversely, if either *t1* fails to satisfy *p1* or *t2* fails to satisfy *p2* then *t3* fails to satisfy *p1∧p2.*

Consider the conjunction "Student *StudentId* is enrolled on course *CourseId* **and** student *StudentId* is called *Name*". The predicates being connected are "Student *StudentId* is enrolled on course *CourseId*" and "Student *StudentId* is called *Name*". We connect two dyadic predicates to form a triadic one—it has only three parameters because the parameter *StudentId* is common to both of the dyadic predicates.

"Student *StudentId* is enrolled on course *CourseId* and student *StudentId* is called *Name*" is of course the intended predicate for the `ENROLMENT` relvar of Figure 1.2 in Chapter 1. Figure 1.2 tells us that `TUPLE { StudentId SID('S1'), Name NAME('Anne'), CourseId CID('C1') }` satisfies that predicate. That being the case, the truth table for conjunction allows us to conclude that it also satisfies both "Student *StudentId* is enrolled on course *CourseId*" and "Student *StudentId* is called *Name*", for if it failed to satisfy either, then the instantiation of their conjunction under that tuple must be false according to that truth table.

Disjunction (OR)

As previously noted, there are plenty of other words (conjunctions) for connecting sentences, and they do not all denote conjunction. For example, "or" is such a word and it denotes *disjunction*, usually denoted in formal treatments by the symbol ∨.

The disjunction of propositions *p* and *q* is true if either *p* is true, or *q* is true, or both are true; otherwise (neither *p* nor *q* is true) false. The truth table for disjunction is shown in Figure 3.3.

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

Figure 3.3: The Truth Table for Disjunction

The definition I have given for disjunction is surely intuitive and obvious. But note that, having previously defined negation and conjunction (in equally intuitive and obvious ways), I could now define disjunction in terms of those two, like this:

$$p \vee q \equiv \neg(\neg p \wedge \neg q)$$

We can use a truth table (Figure 3.4) to prove this equivalence:

p	q	$\neg p$	$\neg q$	$\neg p \wedge \neg q$	$\neg(\neg p \wedge \neg q)$
T	T	F	F	F	T
T	F	F	T	F	T
F	T	T	F	F	T
F	F	T	T	T	F

Figure 3.4: Disjunction in Terms of Negation and Conjunction

We used the truth tables for negation and conjunction to obtain the second, third, and fourth columns. As you can see, the final column is the same as in Figure 3.3.

Like conjunction, disjunction can be applied to predicates in general as well as propositions. Consider the disjunction "Student *StudentId* is enrolled on course *CourseId* **or** student *StudentId* is called *Name*". The predicates being connected are once again "Student *StudentId* is enrolled on course *CourseId*" and "Student *StudentId* is called *Name*". Because the first row of the truth table is the same as the first row of the truth table for conjunction, and because the conjunction of these two predicates is the agreed predicate for `ENROLMENT`, we can conclude that every tuple in `ENROLMENT` satisfies their disjunction too. However many other tuples, not in `ENROLMENT`, also satisfy it. For example, `TUPLE {StudentId SID('S1'), Name NAME('Eve'), CourseId CID('C1')}` satisfies it because it satisfies "Student *StudentId* is enrolled on course *CourseId*" (student S1 isn't called Eve but she *is* enrolled on course C1). For another example, `TUPLE {StudentId SID('S1'), Name NAME('Anne'), CourseId CID('C97')}` satisfies it because it satisfies "Student *StudentId* is called *Name*" (student S1 isn't enrolled on course C97—in fact, as it happens there is no such course—but she *is* called Anne.

You will have realised by now that the cardinality of a relation representing the extension of "Student *StudentId* is enrolled on course *CourseId* or student *StudentId* is called *Name*" is likely to be very high indeed—almost certainly too high to be manageable using a computer. In Chapter 4 you will see that a relational DBMS's support for relations representing disjunctive predicates is subject to a certain restriction, addressing this problem. As with negation, you may also be thinking that relations for disjunctive predicates such as the example at hand would be of little use in practice, in which case you will agree again, when you see the restriction I am referring to, that it is of little or no import.

Conditionals

Consider the sentences shown in Example 3.9.

Example 3.9: conditional sentences

(i) **If** you ask me nicely, **then** I will marry you.
(ii) I will marry you **only if** you ask me nicely
(iii) I will marry you **if and only if** you ask me nicely.

Each denotes a predicate derived from two predicates using a connective of a general kind called conditional. Sentence (i) illustrates use of *logical implication*, usually denoted in formal treatments by the symbol $\rightarrow$. Sentence (i) denotes a true proposition in all situations except when the question is asked nicely but results in refusal. In general, $p \rightarrow q$ ("if p then q") is true except when p is true and q is false.

p	***q***	***p→q***
T	T	T
T	F	F
F	T	T
F	F	T

Figure 3.5: The Truth Table for Logical Implication

As an exercise, you might like to work out the truth table for the expression $\neg p \vee q$. You should find that the final column is identical to the final column of Figure 3.5, confirming the equivalence

$$p \rightarrow q \equiv \neg p \vee q$$

Sentence (ii) of Example 3.9 is merely another way of saying, "If I marry you, then you will have asked me nicely." Sentence (iii) illustrates use of the *biconditional*, also known as *equivalence*, usually denoted in formal treatments by the symbol $\leftrightarrow$. It is true whenever the truth values of "I will marry you" and "You ask me nicely" are identical—either both `TRUE` or both `FALSE`; otherwise it is `FALSE`. It should be clear, then, that $p \leftrightarrow q$ is equivalent to $(p \rightarrow q) \wedge (q \rightarrow p)$, as confirmed by the truth table in Figure 3.6.

p	***q***	***p→q***	***q→p***	**(*p→q*) ∧ (*q→p*)**
T	T	T	T	T
T	F	F	T	F
F	T	T	F	F
F	F	T	T	T

Figure 3.6: $p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$

As you can see, the final column indicates `TRUE` exactly when the first two columns indicate the same truth value.

Quantification

Our final method of deriving a predicate from a predicate does not correspond to anything in propositional calculus and in fact is what distinguishes the *predicate calculus* from propositional calculus.

To quantify something is to state its quantity, to say how many of it there are. Consider the following sentences:

> *x* has been president of the USA.
> 44 people have been president of the USA.

The first denotes a monadic predicate, the second a niladic one (and is either true or false, depending, as it happens, on when it is uttered—for example, it became true in January 2009). The second sentence is in a sense derived from the first by stating how many *x*'s there are such that *x* has been president of the USA.

Quantification doesn't have to be numerically precise. Here are some further examples:

> At least 40 people have been president of the USA.
> Between 40 and 50 people have been president of the USA.
> Nobody has been president of the USA.

That last one *is* precise, of course. It happens to be a false statement, from which it follows that its negation,

> It is not the case that nobody has been president of the USA.

is a true one. But this uses a "double negative" and is just an awkward way of stating

> At least one person has been president of the USA.

Or

> There is a person *x* such that *x* has been president of the USA.

This is derived from "*x* has been president of the USA" by *existential quantification*. Like substitution, quantification reduces the number of parameters: one of the parameters of a given *n*-adic predicate is quantified and the result is an (*n*-1)-adic predicate.

Note carefully that the symbol *x*, denoting a variable, appears twice in that last formulation, and yet it is not a parameter. The *x* in the predicate "*x* has been president of the USA" *is* a parameter, of course. As such it is also referred to sometimes as a *free variable*, when it is then said to be *bound*, by quantification, in "There is a person *x* such that *x* has been president of the USA".

Existential quantification is denoted in formal treatments by the symbol ∃ ("there exists"), as in

> $\exists\ x : x$ has been president of the USA.

As before, the colon can be pronounced "such that".

Having seen how existential quantification can be expressed in a long-winded way using a double negative, now see what happens when we apply a double negative to an existentially quantified predicate, as in the sentence

> It is not the case that somebody doesn't know who is the current president of the USA.

This would be expressed formally as

> $\neg(\exists x : \neg($x$ knows who is the current president of the USA$))$

WHILE YOU WERE SLEEPING...
www.fuqua.duke.edu/whileyouweresleeping
DUKE
THE FUQUA SCHOOL OF BUSINESS

"It is not the case that there exists a person x such that it is not the case that x knows who is the current president of the USA." You have perhaps worked out by now that all we are saying is, "Everybody knows who is the current president of the USA." Here we are using *universal quantification*—stating that something is true of everything that is under consideration. Universal quantification is denoted in formal treatments by the symbol ∀ ("for all"), as in

$\forall x$: x knows who is the current president of the USA

(The colon here clearly cannot be pronounced "such that" but is kept for symmetry with existential quantification.)

That concludes my digression into predicate logic to explain how relations are supposed to be interpreted. In the next chapter I show how the operators of the relational algebra are used to derive relations from relations, relating these to the methods I have described for deriving predicates from predicates. You might like to try the exercises that follow before moving on to that chapter.

EXERCISES

Consider again the relation shown as the current value of ENROLMENT in Figure 1.2, repeated here for convenience:

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

For each of the following propositions, state whether it is true or false, basing your conclusions on this relation:

1. There exists a course *CourseId* such that some student named Anne is enrolled on *CourseId.*
2. Every student with StudentId S1 who is enrolled on some course is named Anne.
3. Every student who is enrolled on course C4 is named Anne.
4. Some student who is enrolled on course C4 is named Anne.
5. There are exactly 5 students who are enrolled on some course.
6. It is not the case that there is no course on which no student who is enrolled on some course but is not named Boris is not enrolled.
7. There are exactly 10 pairs of StudentIds (*SID1, SID2*) such that there is some course on which student *SID1* is enrolled and student *SID2* is enrolled.
8. There are exactly 3 pairs of StudentIds (*SID1, SID2*) such that there is some course on which student *SID1* is enrolled and student *SID2* is enrolled.
9. If a student named Eve is enrolled on course C1, then student S1 is named Adam.
10. If student S1 is named Anne, then S1 is enrolled on course C2.

4 Relational Algebra— The Foundation

4.1 Introduction

This chapter introduces you to *relational operators*. A relational operator is an operator that takes one or more relations as operands and produces a relation as a result. The relational operators described in this chapter constitute a basic set that is considered to be sufficient for *relational completeness*. Such a set of operators is called a *relational algebra*. My use of the word "basic" suggests that this set of operators would be *necessary*, as well as sufficient, for relational completeness. That will do as a working assumption for now, but we shall eventually see that there is actually a tiny element of superfluity.

Sometimes that term, relational algebra, is used with the definite article: *the* relational algebra, even though several minor variations exist in the literature. Indeed, the term relational completeness is sometimes defined with reference to "the" relational algebra—a language is deemed relationally complete if it supports, directly or indirectly, all of the operators of "that" algebra. Suffice it here to say that the operators described in this chapter meet the criteria for constituting a complete relational algebra, and their manifestation in **Tutorial D** makes that language relationally complete.

We start by looking at a particularly important relational operator, named `JOIN` (see Section **4.4 JOIN and AND** for details). Consider the relations depicted in Figure 4.1, the current values of relvars named `IS_CALLED` and `IS_ENROLLED_ON`.

IS_CALLED

StudentId	Name
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

IS_ENROLLED_ON

StudentId	CourseId
S1	C1
S1	C2
S2	C1
S3	C3
S4	C1

Figure 4.1: IS_CALLED and IS_ENROLLED_ON

Notice that nearly all of the data shown Figure 4.1 appears also in Chapter 1, Figure 1.2, the current value of the variable ENROLMENT. In fact, the relation that is the current value of ENROLMENT can easily be derived from the current values of IS_CALLED and IS_ENROLLED_ON by invoking JOIN, as shown in Example 4.1.

Example 4.1: Joining IS_CALLED and IS_ENROLLED_ON

```
IS_CALLED JOIN IS_ENROLLED_ON
```

This is an example of a *relational expression*. Relational expressions in **Tutorial D** include:

- invocations of the relation selector, RELATION, defined in Chapter 2;
- invocations of relational operators;
- names of relation variables such as IS_CALLED and IS_ENROLLED_ON; and
- the names TABLE_DEE and TABLE_DUM, defined later in the present chapter.

Note that operands of relational operators are denoted by relational expressions.

Figure 4.2 shows the result of evaluating IS_CALLED **JOIN** IS_ENROLLED_ON. You can see that it depicts exactly the same relation as the current value of ENROLMENT shown in Figure 1.2.

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

Figure 4.2: The result of joining IS_CALLED with IS_ENROLLED_ON

The predicate for ENROLMENT is "Student *StudentId* is called *Name* and is enrolled on course *CourseId*". The predicate for IS_CALLED is "Student *StudentId* is called *Name*" and the predicate for IS_ENROLLED_ON is "Student *StudentId* is enrolled on course *CourseId*". Notice that the predicate for ENROLMENT is the conjunction (AND) of the predicates for IS_CALLED and IS_ENROLLED_ON. In general, if *p1* and *p2* are predicates represented by relations *r1* and *r2*, respectively, then the **Tutorial D** expression *r1* JOIN *r2* denotes the relation corresponding to the conjunction *p1* AND *p2* (written as $p1 \wedge p2$ in mathematical notation).

By studying this little example you might be able to work out for yourself how JOIN works, in general, before I explain it in detail. If you try that exercise, there are a couple of points of difference you should take careful note of in Figures 4.1 and 4.2:

a) The only fact depicted in Figure 4.1 but not in Figure 4.2 is that student S5 is called Boris. Student S5 doesn't appear in Figure 4.2 because that student isn't enrolled on any courses at all. Therefore there is no course identifier—no value of type CID—that satisfies the predicate "Student S5 is called Boris and is enrolled on course *CourseId*".
b) In Figure 4.1 the fact that student S1 is called Anne is shown just once, whereas in Figure 4.2 it is shown twice.

JOIN is an operator that, when invoked, operates on two relations and returns a relation. A relational operator is one that, when invoked, operates on one or more given relations and returns a relation. A relational algebra is a set of such operators—in mathematical parlance it (the algebra) is *closed over* relations. The term *closure* refers to the property of a set of operators whereby the results are of the same type as the operands. For example, the familiar operators of arithmetic—"+", "-", and so on—are closed over numbers. This chapter and the next describe the set of relational operators defined for **Tutorial D**, giving for each one the notation used in **Tutorial D** for invoking it. The ones described in this chapter are just those needed for relational completeness. In Chapter 5, "Building on The Foundation", I describe some further operators defined in **Tutorial D** to illustrate some of the additional kinds of things that are needed to make a relational database language more useful for practical purposes.

The aforementioned property of closure allows us to write relational expressions of arbitrary complexity, because an invocation of a relational operator, denoting, as it does, a relation, can be used to denote an argument to another invocation. Thus, invocations can be nested inside each other to any degree of complexity. (Recall that the operators of arithmetic allow us to write numerical expressions of arbitrary complexity in similar fashion.)

In Chapter 3 we saw how a relation might represent the extension of some predicate. We also saw the various ways in which predicates can be derived from predicates, using logical connectives and quantifiers. For each such method we will discover a corresponding relational operator that can be used to derive the relation representing the extension of a predicate derived by that method, just as I have shown for JOIN in Example 4.1.

4.2 Relations and Predicates

Recall from Chapter 3 *how* a relation represents the extension of a predicate:

> A relation represents the extension of some *n*-adic predicate *P* by having a body consisting of every *n*-tuple that satisfies *P*.

The current value of `IS_CALLED` shown in Figure 4.1 tells us that the extension of "Student *StudentId* is called *Name*" is (currently) the set {Student S1 is called Anne, Student S2 is called Boris, Student S3 is called Cindy, Student S4 is called Devinder, Student S5 is called Boris}, each element of which is an instantiation of the predicate that happens to be true. From the given definition it follows that every other instantiation of that predicate is false and therefore does not appear in its extension. The assumption that the body of a relation consists of *every* tuple that satisfies the relation's predicate is called the *closed world assumption*. This assumption underpins the operators of the relational algebra. Under the *open world assumption*, every tuple that's in the relation does represent a true proposition but it is possible for tuples that satisfy the predicate not to appear in the relation. It would not be possible to devise relational operators based on that assumption, for unless every instantiation of the predicate were false it would be an arbitrary choice as to which tuples, if any, are to be included.

4.3 Relational Operators and Logical Operators

The operators of the relational algebra are nearly all relational counterparts of the logical connectives `AND`, `OR`, and `NOT`, and existential quantification. Each relational operator, when invoked, yields a relation, which can be interpreted as the extension of some predicate. Because relations are used to represent predicates, it makes sense for relational operators to be counterparts of logical operators. Example 4.1 illustrates the use of `JOIN` as the relational counterpart of the logical operator `AND` to give us the relation representing the predicate

Student *StudentId* is called *Name* `AND` *StudentId* is enrolled on course *CourseId*.

where the predicates for `IS_CALLED` and `IS_ENROLLED_ON` are connected by `AND`.

We will meet other examples, enabling us to obtain relations representing predicates such as

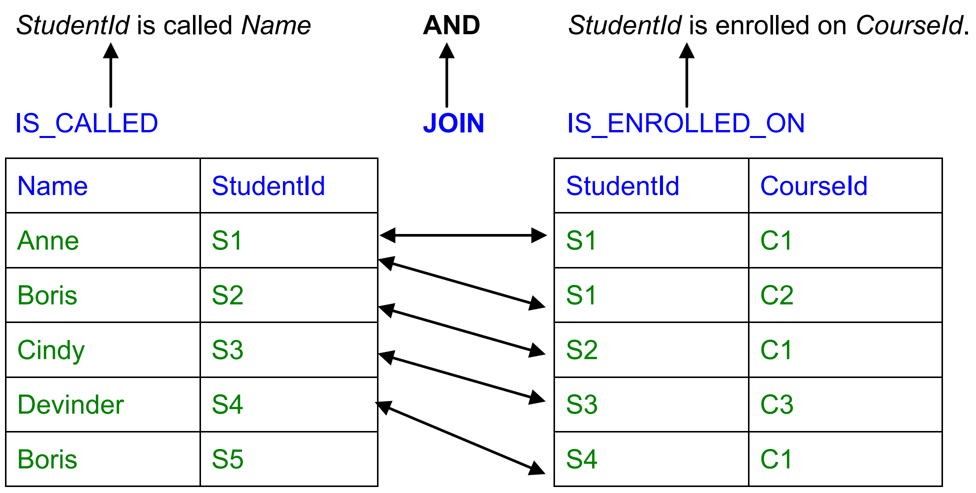
- Student *StudentId* is enrolled **on some course**.
  Here the predicate for `IS_ENROLLED_ON` is subject to existential quantification on *CourseId*.
- Student *StudentId* is enrolled on course *CourseId* AND *StudentId* is **NOT** called Devinder.
  Here the predicate for `IS_ENROLLED_ON` is connected by `AND` to the negation of the monadic predicate formed by substituting Devinder for *Name* in the predicate for `IS_CALLED`.
- Student *StudentId* is NOT enrolled on any course **OR** *StudentId* is called Boris.
  Here the negation of the predicate resulting from existential quantification of the predicate for `IS_ENROLLED` is connected by `OR` to the predicate formed by substituting Boris for *Name* in the predicate for `IS_CALLED`.

and more. Descriptions of relational operators now follow, starting with `JOIN`. In most cases the section heading shows the logical operator to which the given relational operator corresponds. You will see that in at least one case, `AND`, we have several distinct relational operators and I will explain why this is so.

4.4 JOIN and AND

Figure 4.3 shows how Example 4.1 "works". I have changed the order of the columns in the `IS_CALLED` table (you know that makes no difference) in order to place side by side the attributes that are both named `StudentId`. The upward arrows show which bits of the relational expression correspond to which bits of the predicate.

The arrows connecting rows in the tables show which combinations of operand tuples represent true instantiations of the predicate. Note how a tuple on the left *matches* a tuple on the right if and only if the two tuples have the same `StudentId` value. This concept of matching tuples is at play in several of the other operators too. Tuples *t1* and *t2* are said to match, or be matching, if and only if, for each common attribute *c*, they have the same value for *c*, where a common attribute is one that has the same name in both *t1* and *t2*.


StudentId is called *Name*
AND
StudentId is enrolled on *CourseId*.
IS_CALLED
JOIN
IS_ENROLLED_ON

Name	StudentId
Anne	S1
Boris	S2
Cindy	S3
Devinder	S4
Boris	S5

StudentId	CourseId
S1	C1
S1	C2
S2	C1
S3	C3
S4	C1

Figure 4.3: Joining IS_CALLED with IS_ENROLLED_ON

Common attributes have to be of the same type as well as having the same name. For consider, if that were not the case in our `JOIN` example—suppose, for example, that `StudentId` in `IS_CALLED` were of type `SID` and `StudentId` in `IS_ENROLLED_ON` were of type `CHAR`—then:

a) What would be the type, `SID` or `CHAR`, of the single `StudentId` attribute in the result of the `JOIN`?
b) How could the DBMS determine whether a given pair of tuples match on `StudentId`? To be comparable at all, the operands of equals comparison must be of the same type.

Note very carefully that the result does have only one `StudentId` attribute, even though the corresponding parameter appears twice in the predicate. Multiple appearances of the same parameter in a predicate are always taken to stand for the same thing. Here, if we substitute S1 for one of the `StudentId`s, then we must also substitute S1 for the other. That is why we have only one `StudentId` in the resulting relation. `StudentId` is a *common attribute* of *r1* and *r2*. In general, there can be any number of common attributes; and there can even be no common attributes at all.

Definition of JOIN

Here is a formal definition of our first relational operator, `JOIN`:

Let *s* = *r1* **JOIN** *r2*. Then:

- The heading *Hs* of *s* is the union of the headings of *r1* and *r2*, which must be a heading (otherwise, *r1* JOIN *r2* is undefined).
- The body of *s* consists of those tuples having heading *Hs* that can be formed by taking the union of *t1* and *t2*, where *t1* is a tuple of *r1* and *t2* is a tuple of *r2*.

If both *r1* and *r2* have an attribute of the same name but not the same type, then the union of their headings is not a heading and *r1* JOIN *r2* is undefined.

We can take the union of two headings because headings are sets. In particular, a heading is a set of attribute name/type pairs. If two headings have different types for some attribute of the same name, then their union is a set (of course) but is not a heading—because a heading cannot have more than one type paired with the same name.

We can take the union of two tuples because tuples are sets. In particular, a tuple is a set of attribute name/value pairs. If two tuples have different values for some common attribute, then their union is a set (of course) but is not a tuple—because a tuple cannot have more than one value paired with the same name.

Note that if either operand is empty (its body is the empty set), then so is the result.

Interesting properties of `JOIN`

`JOIN` is both *commutative* and *associative*. Commutativity (of a dyadic operator) means that the order of operands is insignificant. That is to say, *r1* `JOIN` *r2* is equivalent to *r2* `JOIN` *r1*. Associativity means that (*r1* `JOIN` *r2*) `JOIN` *r3* is equivalent to *r1* `JOIN` (*r2* `JOIN` *r3*)—if we wish to join three or more relations together, then we can join them in any order, so to speak. Of course it is no mere coincidence that logical `AND` is also both commutative and associative.

Properties such as these not only save us a certain amount of thinking when formulating expressions; they also help the optimizer to find alternative formulations that might perform better than a straightforward implementation of the one written. **Tutorial D** takes further advantage of these properties by supporting an alternative syntax for invoking `JOIN`, using *prefix* notation instead of the *infix* notation I have already shown you. In prefix notation the operator name is followed by a list of argument expressions in braces. When the operator is associative, we can have any number of arguments:

```
JOIN { r1, r2, ... }
```

When there is just one argument, *r1,* the result is *r1.* I hope you are wondering what happens if there are no arguments at all—asking if `JOIN { }` is defined. Well, it is!—but I have to defer the explanation until later, for a reason you will understand when I do so.

As well as being commutative and associative, `JOIN` is *idempotent*. Let *r* be any relation. Then *r* `JOIN` *r* = *r.* A dyadic operator is idempotent if, when its operands are the same value, it yields that value. Again, it is no mere coincidence that logical `AND` is idempotent: $p \land p$ always has the same truth value as *p.*

There is one further interesting property of `JOIN`, described later, in Section 4.6 of this chapter.

Two special cases of `JOIN`

There are two extreme cases concerning common attributes: the case when all attributes are common to both operands and the case where none of them are.

If all attributes of *r1* and *r2* are common, then the body of *r1* `JOIN` *r2* is the set-theory *intersection* of the body of *r1* and the body of *r2.* For that reason their join is sometimes called intersection. In fact, **Tutorial D** allows you to use the key word `INTERSECT` in place of `JOIN` in this special case only. If you do so, you are in effect telling the system that you expect the operands to have all attributes in common, and you might be grateful if the system rejects your invocation (at compile time) when they don't.

If no attributes of *r1* and *r2* are common, then every pair of tuples, *t1* from *r1* and *t2* from *r2*, is such that their union is a tuple and so appears in the body of *r1* `JOIN` *r2*. In mathematical terms we have the same combinations of tuples as would appear in the *Cartesian product* of the two bodies (the analogy is a little loose, because in the Cartesian product the paired elements remain as a pair, whereas `JOIN` combines them to form a single tuple). For that reason their join is sometimes called their product and **Tutorial D** allows you use the key word `TIMES` in place of `JOIN` in this special case only. If you do so, you are in effect telling the system that you expect the operands to have no attributes in common, and you might be grateful if the system rejects your invocation (at compile time) when they don't.

Now, sometimes, when joining relations, we want attributes *a1* of *r1* and *a2* of *r2* to be used for determining matching tuples even though *a1* and *a2* do not have the same name. Conversely, we sometimes want common attributes not to take part in the matching process. In such cases we cannot simply take the result of *r1* JOIN *r2*. We have to somehow "change" some attribute names before we do the join; and that brings me to the next operator, RENAME.

4.5 RENAME

Although I offer no logical counterpart for RENAME, consider that one predicate can be derived from another predicate simply by changing one or more of its parameter names. For example, from the predicate for IS_CALLED, "Student *StudentId* is called *Name*", we could derive the different predicate "Student *Sid* is called *Name*". Those two predicates have identical extensions and in fact have identical *in*tensions too; but they aren't represented by exactly the same relations, these being the two relations depicted in Figure 4.4.

StudentId	Name
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

Sid	Name
S1	Anne
S2	Boris
S3	Cindy
S4	Devinder
S5	Boris

Figure 4.4: Relations differing only in an attribute name

The one on the left is the current value of IS_CALLED. The one on the right is the same relation except that the attribute name Sid is used in place of StudentId. In some circumstances we might want to derive the relation on the right from the one on the left and use the result in some operation such as a JOIN. That's what the RENAME operator is for, and Example 4.2 shows how to use it to obtain the relation on the right.

Example 4.2: Renaming an attribute

```
IS_CALLED RENAME { StudentId AS Sid }
```

The expression enclosed in parentheses—which is not a relational expression—is called a renaming. In general there can be any number of renamings, separated by commas.

Definition of **RENAME**

Let *s* = *r* **RENAME** { *a1* **AS** *b1*, …, *an* **AS** *bn* }. Then:

- The heading of *s* is the heading of *r* except that the attribute named *ai*, of type *ti*, is replaced by an attribute named *bi*, also of type *ti* ($0 \leqslant i \leqslant n$).
- The body of *s* consists of the tuples of *r* except that in each tuple attribute *ai* with value υ is replaced by attribute *b* with value υ.

Obviously, the result must be such that no two attributes have the same name; otherwise, it wouldn't be a relation. Note that when more than one attribute is being renamed, the renamings are considered to take place simultaneously, so to speak. Thus, the example `R RENAME { A AS B, B AS A }` swaps the names of those two attributes.

Syntax note

When describing syntax we use the convenient term *commalist* for a list of items separated by commas. Thus we can say that an invocation of `RENAME` consists of a relation expression followed by the key word itself, followed in turn by a renaming commalist enclosed in parentheses.

In **Tutorial D**, wherever the syntax requires a commalist, that commalist is permitted to be empty. Thus, for example, `R RENAME { }` is a legal expression. (Its value is equal to `R`.)

Please do not confuse the `RENAME` operator with one that might be used for changing the name of an attribute of a relvar. That might be a very useful tool for database designers but in this chapter we are dealing only with *read-only* operators that yield relations. None of them has any effect on the database's contents or definition.

Unfortunately, E.F. Codd did not foresee the need for a `RENAME` operator and so it is missing from some accounts of relational algebra that you may come across.

Using RENAME in combination with JOIN

Suppose we wish to discover pairs of students who have the same name. The result must be a relation with three attributes: two for the student identifiers and one for the name those two students share. All the data we need for that ternary relation is in the binary relvar `IS_CALLED`. The predicate for `IS_CALLED` is "Student *StudentId* is called *Name*". What might be a predicate for our desired result? Obviously we cannot just connect "Student *StudentId* is called *Name*" to itself using `AND`, because multiple appearances of the same parameter must all represent the same value. So we must use two different parameter names for the two student identifiers. Example 4.3 shows a suitable predicate, followed by a relational expression that uses `RENAME` (twice) and `JOIN` to denote the required relation.

Example 4.3: Renaming and joining

Student *Sid1* is called *Name* and so is student *Sid2*

```
( IS_CALLED RENAME { StudentId AS Sid1 } ) JOIN
( IS_CALLED RENAME { StudentId AS Sid2 } )
```

Note that `Name` is the only common attribute for the `JOIN`. Now you can begin to see how relational operators can be used to construct expressions of unlimited complexity. Unfortunately, though, the result obtained from this expression, shown in Figure 4.5, isn't entirely satisfactory.

Sid1	Name	Sid2
S1	Anne	S1
S2	Boris	S2
S2	Boris	S5
S5	Boris	S2
S5	Boris	S5
S3	Cindy	S3
S4	Devinder	S4

Figure 4.5: Result of Example 4.3

You didn't really want to be told that Anne has the same name as herself, nor that students S5 and S2 share the same name (Boris) as well as students S2 and S5 sharing that name! We need to look at some more operators before we can begin to address those little problems. The next one is called *projection*.

4.6 Projection and Existential Quantification

Suppose we need to obtain the student identifiers of all the students who are enrolled on some course. Even though that result perhaps isn't very interesting in itself, we might need it as part of some more interesting query. The relation we require would represent the predicate derived from the predicate for `IS_ENROLLED_ON` by existential quantification of *CourseId:*

Student *StudentId* is enrolled on some course.

or, more formally

There exists a course *CourseId* such that student *StudentId* is enrolled on *CourseId*.

Example 4.4 shows how to obtain the relation representing this predicate, using *projection*.

Example 4.4: Projection

Student *StudentId* is enrolled on some course.

```
IS_ENROLLED_ON { StudentId }
```

Points to note:

- Like `RENAME`, projection is monadic (it operates on just a single relation, in this case the current value of `IS_ENROLLED_ON`).
- **Tutorial D** uses no key word for projection. You just write a commalist of attribute names (a list of one in the example), enclosed in braces, after the expression denoting the single relation operand.
- The braces indicate that the order of attribute names in the given list is insignificant. Indeed, here it denotes a set (in the example, a set with just one element).
- The attributes named in braces are exactly those that remain if we remove the parameters that are existentially quantified in the predicate. Sometimes it is more convenient to name the attributes to be excluded rather than the remaining ones. With this in mind, **Tutorial D** supports an alternative formulation, using `ALL BUT`. Thus Example 4.4 could have been expressed like this instead:
  ```
  IS_ENROLLED_ON { ALL BUT CourseId }
  ```

The result is shown in Figure 4.6.

StudentId
S1
S2
S3
S4

Figure 4.6: Result of Example 4.4

Points to note:

- The degree is one, that being the number of attributes specified in the projection.
- Every `StudentId` value in the result is a `StudentId` value that appears in the operand relation, `IS_ENROLLED_ON` (and no other `StudentId` value appears in the result).
- No `StudentId` value appears more than once in the result, even though S1 appears twice in the operand. The body of a relation is a set and there is no sense in which the same element can appear more than once in a set.

Definition of projection

Let *s* = *r* { *a1*, …, *an* }
 = *r* {ALL BUT *b1*, …, *bm* }
where the sets { *a1*, …, *an* } and { *b1*, …, *bm*} are disjoint subsets of the heading of *r*, whose union contains every attribute of *r*.

Then:

- The heading of *s* is the subset of the heading of *r* given by { *a1*, …, *an* }.
- The body of *s* consists of each tuple that can be formed from a tuple of *r* by removing from it the attributes named *b1*, …, *bm*.

Note that the cardinality of *s* can be less than that of *r* but cannot be more than that of *r*. Now look at Figure 4.7, showing the result of

```
( ( IS_CALLED RENAME { StudentId AS Sid1 } ) JOIN
 ( IS_CALLED RENAME { StudentId AS Sid2 } ) ) { ALL BUT Name }
```

Sid1	Sid2
S1	S1
S2	S2
S2	S5
S5	S2
S5	S5
S3	S3
S4	S4

Figure 4.7: A projection of the relation shown in Figure 4.5

As you can see, the figure shows pairs of student identifiers that identify pairs of students having the same name. Its predicate is "Student *Sid1* has the same name as student *Sid2*". There is something I want to say about this predicate although it amounts to something of a digression at this point. Because it is true that every student has the same name as himself or herself, we say that the relation for this predicate is *reflexive*. Because it is also true that if student *x* has the same name as student *y*, then student *y* has the same name as student *x*, we say that the relation for this predicate is *symmetric*. In general, binary relations with attributes of the same type are said to be *recursive*. Clearly a relation that is reflexive must be recursive, for if *x* and *y* denote the same thing, they must be of the same type. A relation that is symmetric must also be recursive, for otherwise it would not be possible to interchange *x* and *y* (if *x* is a student id and *y* is a course id, for example).

How ENROLMENT was split

Our current examples, `IS_CALLED` and `IS_ENROLLED_ON`, when joined yield the current value of `ENROLMENT`, as we have already seen. Now, we cannot exactly reverse that process—obtain the current values of `IS_CALLED` and `IS_ENROLLED_ON` from `ENROLMENT`—but we can, by using projection, obtain the current value of `IS_ENROLLED_ON`, and we can obtain a subset of the current value of `IS_CALLED` (student S5 would be missing).

Suppose we have only `ENROLMENT`, but we suddenly realize that we need to record the names of students, such as student S5, who aren't yet enrolled on any courses. We decide to address that problem by using the two relvars, `IS_CALLED` and `IS_ENROLLED_ON`, in place of `ENROLMENT`. Example 4.5 shows one way of achieving that redesign in **Tutorial D**, using projection and a certain useful shortcut you can use in relvar declarations. (You may wish to return to Chapter 2, Section 2.11, to refresh your memory on relvar declarations in **Tutorial D**.)

Example 4.5: Splitting ENROLMENT

```
VAR IS_CALLED BASE
INIT (ENROLMENT { StudentId, Name })
KEY { StudentId } ;

VAR IS_ENROLLED_ON BASE
INIT (ENROLMENT { StudentId, CourseId })
KEY { StudentId, CourseId } ;

DROP VAR ENROLMENT ;
```

Explanation 4.5:

- **`VAR IS_CALLED BASE`** announces that what follows defines a database relvar named `IS_CALLED`.
- **`INIT (ENROLMENT { StudentId, Name })`** specifies that the variable is to be immediately assigned the result of projecting the current value of `ENROLMENT` over `StudentId` and `Name`. It also implies that the declared type of the specified expression, `ENROLMENT { StudentId, Name }`, is the declared type of the variable. This is the useful shortcut I mentioned—when `INIT` is used there is no need to spell out the type.
- **`KEY { StudentId }`** specifies a constraint to the effect that no two distinct tuples having the same `StudentId` value can ever appear simultaneously in `IS_CALLED`. (In **Tutorial D** the `KEY` specification must come *after* the type specification, including the type implicitly specified by `INIT`.)

- **`VAR IS_ENROLLED_ON`** up to the next semicolon is a similar relvar declaration for `IS_ENROLLED_ON`. The `KEY` specification, `KEY { StudentId, CourseId }`, specifies a constraint to the effect that no two distinct tuples having the same `StudentId` value and the same `CourseId` value can ever appear simultaneously in `IS_ENROLLED_ON`. It is superfluous, really, because those are the only two attributes and it is *never* possible for the same tuple to appear more than once in the body of a relation, by definition. However, **Tutorial D** requires at least one `KEY` specification to be included in a relvar declaration.
- **DROP VAR ENROLMENT** destroys the variable we have no further use for.

Of course, our revised database does not yet include any information about student S5, named Boris, but that information wasn't present in `ENROLMENT`—and nor could it have been, until S5 became enrolled on something. Now, however, we can record S5's name immediately, by adding the appropriate tuple to `IS_CALLED`.

Two special cases of projection

In **Tutorial D** syntax, wherever a commalist of items is required it is permissible for that list to be empty unless it is explicitly stated to the contrary. At the time of writing there are no exceptions; therefore the following two expressions must both be legal:

```
r { ALL BUT }
r { }
```

where *r* denotes a relation.

The first should come as no surprise—it obviously results in *r* itself—but the second does surprise most students at first, for it appears to denote a relation with no attributes at all—a relation of degree zero. And indeed it does, and indeed there are such relations!—but only two. The two relations have been given the pet names `TABLE_DEE` and `TABLE_DUM` and these names, which were proposed by the present author in a journal article published in 1988, are available in **Tutorial D**, which allows them to be abbreviated to just `DEE` and `DUM`.

`TABLE_DEE` is a name for the relation `RELATION { TUPLE { } }`—the relation of degree zero and cardinality one. There is only one tuple of degree zero, so that has to be the only tuple of `TABLE_DEE`. Clearly there cannot be a relation of degree zero and cardinality greater than one, for then we would have the same tuple appearing more than once in the body of a relation and that, as we have seen, cannot be. So `TABLE_DUM` must be the empty relation of degree zero: `RELATION { } { }` (the first `{ }` specifies the empty heading, the second the empty body).

A predicate represented by a relation of degree zero is niladic (has no parameters). In other words, it must be a proposition, *p*. If `TABLE_DEE` represents *p*, then *p* is true; otherwise `TABLE_DUM` represents *p* and *p* is false. People often ask, "What purpose can relations of degree zero possibly serve? They seem to be of little or no value." The answer is that they represent answers to queries of the form "Is it true that…?" or "Are there any…?" where the answer is just "yes" or "no". For example, "Is it true that student S1 is enrolled on course C3?", and "Are there any students enrolled on course C1?"

Now that you have met `TABLE_DEE` I can show you one more interesting property of `JOIN`. If *r* is, as usual, a relation, what is the result of *r* `JOIN TABLE_DEE`? Even if an answer springs to mind immediately I suggest you work this out for yourself from the definition of `JOIN`—and perhaps verify your conclusion using *Rel*.

If a value *i* exists such that whenever *i* is one of the operands of a dyadic operator the result of invoking that operator is the other operand, then *i* is said to be an *identity* value under that operator. Think of the number 0 under addition, for example, and the number 1 under multiplication. `TABLE_DEE` is the identity value under `JOIN`.

Now, consider an operator that is commutative and associative, as are numerical addition and relational JOIN. As we have seen, a language can support *n*-adic versions of such operators: we can take the sum of any number of numbers and we can take the join of any number of relations—even just one or none at all. The sum of just one number is that number and the join of just one relation is that relation. The sum of no numbers is zero, because zero is the identity value under addition. The join of no relations is TABLE_DEE.

So far I have shown you relational counterparts of AND and existential quantification. Eventually you will see counterparts of OR and NOT too but we haven't finished with AND yet, for JOIN turns out not to be suitable for certain common special cases of AND. The next operator addresses one of those cases and is called *restriction*.

4.7 Restriction and AND

Here is a predicate that we can derive from the predicate for IS_CALLED by substitution of one of its parameters:

Student *StudentId* is called Boris.

The relation representing that can be obtained quite easily using JOIN and projection, noting that the given predicate is equivalent to the more elaborate

There exists a name *Name* such that student *StudentId* is called *Name* and *Name* is Boris.

Again the only parameter is *StudentId*, because *Name* is quantified. The relation is denoted by the expression shown in Example 4.6.

Example 4.6: JOIN and projection

```
( IS_CALLED JOIN RELATION { TUPLE { Name NAME ( 'Boris' ) } } )
{ StudentId }
```

Restriction, invoked using the key word WHERE, gives us an alternative and perhaps more intuitive formulation for the first line of Example 4.6, shown in Example 4.7.

Example 4.7: Restriction

```
( IS_CALLED WHERE Name = NAME ( 'Boris' ) )
```

Here the word WHERE is preceded by a relational expression and followed by a *condition*, Name = NAME ('Boris'). Each tuple of the specified relation is tested to see if it satisfies the given condition. Those that do satisfy it, and only those tuples, appear in the result.

Notice that the expression `Name = NAME ( 'Boris' )` is not one that can be evaluated outside of the context in which it appears. Its evaluation depends on the existence of a tuple to provide the value of the attribute `Name`. We shall refer to such expressions as *open* expressions. Unsurprisingly, we shall refer to expressions that *can* be evaluated independently of their context as *closed* expressions. We shall soon see that restriction isn't the only context in which open expressions can appear.

Of course the `WHERE` condition doesn't *have* to be an open expression; but if it is closed, then its value is independent of the tuples of the operand relation and is therefore the same for each tuple. As a consequence, the result of a restriction whose condition is a closed expression is either the input relation (the `WHERE` condition evaluates to `TRUE`) or empty (it evaluates to `FALSE`).

Now, the reason why the example at hand can be formulated using `JOIN` instead of `WHERE` lies in the fact that the restriction condition is, very specifically, an "equals" comparison of an attribute with a literal. The literal in question can be "wrapped up", so to speak, as a relation literal that can be used as an operand of `JOIN`. If the condition is less restrictive, the required relation literal might be far too large to be written down. Suppose, for example, that we wish to find all the students whose names begin with the letter "B". Then we have to replace the relation literal in Example 4.6 by one that includes a tuple literal for every value of type `NAME` that begins with the letter "B". That is out of the question. But, using the `STARTS_WITH` operator from *Rel*'s `OperatorsChar.d` (recall that we used this operator in the definition of type `SID` in Chapter 2), the task becomes very easy using `WHERE`, as shown in Example 4.8.

Example 4.8: A more useful restriction

```
IS_CALLED WHERE STARTS_WITH(THE_C(Name), 'B')
```

Definition of restriction

Let *s* = *r* **WHERE** *c*, where *c* is a possibly open truth-valued expression denoting a condition on attributes of *r*. Then:

- The heading of *s* is the heading of *r*.
- The body of *s* consists of those tuples of *r* for which the condition *c* evaluates to TRUE.

So the body of *s* is a subset of the body of *r*. Note that the result of *r* `WHERE TRUE` is *r* and that of *r* `WHERE FALSE` is the empty relation of the same type as *r*. In fact, whenever the specified condition *c* is a closed expression the result of *r* `WHERE` *c* is empty when *c* evaluates to `FALSE` and is otherwise equal to *r*. So closed expressions are rarely useful and in practice *c* is nearly always an open expression.

Now let us return to Example 4.3, finding pairs of students who have the same name. It was annoying to find those tuples that pair students with themselves. Now we know how those can be eliminated:

```
( ( IS_CALLED RENAME { StudentId AS Sid1 } )
  JOIN
  ( IS_CALLED RENAME { StudentId AS Sid2 } )
WHERE NOT (Sid1 = Sid2) ) { Sid1, Sid2 }
```

which yields the relation shown in Figure 4.8.

Sid1	Sid2
S2	S5
S5	S2

Figure 4.8: An improvement on Figure 4.7

If we are still annoyed by seeing two different students paired together twice, we might be able to address that problem too, if a comparison operator such as "less than" is available on values of type SID:

```
( ( IS_CALLED RENAME { StudentId AS Sid1 } )
  JOIN
  ( IS_CALLED RENAME { StudentId AS Sid2 } )
WHERE Sid1 < Sid2 ) { Sid1, Sid2 }
```

Assuming that the value SID('S2') precedes SID('S5') in the ordering defined for values of type SID, this would yield the singleton relation shown in Figure 4.9.

Sid1	Sid2
S2	S5

Figure 4.9: A further improvement on Figure 4.7

Now please look again at Example 4.8. It involves computation of the first letter of every student's name, for testing in the WHERE condition. Sometimes we wish to use such computations to obtain values that are to appear as attribute values in some relation. Suppose, for a rather unreal example, that we wish to obtain a relation showing not only the student identifier and name of each student, but also the first letters of their names. Then we will need our next operator, also related to AND, called *extension* (a slightly unfortunate name, perhaps—not to be confused with extensions of predicates as defined in Chapter 3!).

4.8 Extension and AND

Consider, then, the predicate

Student *StudentId* is called *Name* and *Name* begins with the letter *Initial*.

We have AND connecting the predicate for IS_CALLED with "*Name* begins with the letter *Initial*". As with Example 4.8, it is not feasible to write down a relational literal representing "*Name* begins with the letter *Initial*", so we cannot feasibly use JOIN.

Here is a **Tutorial D** formulation using extension:

```
EXTEND IS_CALLED : { Initial := FirstLetter ( Name ) }
```

Here FirstLetter (Name) is an open expression—the expression needs to be evaluated against a tuple that provides an attribute value for Name. The expression is evaluated for each tuple *t* of IS_CALLED, yielding the tuple formed by "extending" *t* by the attribute Initial having the value of that open expression.

The result is the relation shown in Figure 4.10.

StudentId	Name	Initial
S1	Anne	A
S2	Boris	B
S3	Cindy	C
S4	Devinder	D
S5	Boris	B

Figure 4.10: An extension of IS_CALLED

The construct `Initial := FirstLetter ( Name )` is called an "extend addition". In general, there can be any number of extend additions, including none at all (in which case the invocation returns its input relation). The extend additions are considered to be evaluated in order from left to right, so that a subsequent extend addition may use an open expression that refers to an attribute "added" by an earlier one.

Definition of extension

Let *s* = **`EXTEND`** *r* : { *a* := *exp* }

where relation *r* does not have an attribute named *a* and *exp* is a possibly open expression. Then:

- Let *T* be the declared type of *exp* and let *Hr* be the heading of *r*. The heading of *s* is *Hr* `UNION` { *a T* }.
- The body of *s* consists of tuples formed from those of *r* by adding the attribute *a* of type *T* with value *exp*.

`EXTEND` *r* : { *a1* := *exp-1*, …, *an* := *exp-n* }, where n ⩾ 0, is equivalent to

`EXTEND` (… (`EXTEND` *r* : { *a1* := *exp-1* } …) : { *an* := *exp-n* })

Points to note:

- In the special case where the commalist of extend additions is empty, the input relation is returned. In other words, `EXTEND` *r* : { } = *r*.
- If *ai* (0⩽i⩽n) is identical to the name of some attribute of *r*, then that attribute is effectively replaced (in which case "extension" is something of a misnomer!).
- The cardinality of *s* is equal to the cardinality of *r*. The degree of *s* the degree of *r* plus one for each *ai* that is not identical to the name of some attribute of *r*.
- If a closed expression is used in an extend addition, then it will have the same value for each tuple of the input relation.

Now we have finished with AND and we can move to OR, where the corresponding relational operator is UNION.

4.9 UNION and OR

Recall that the relational expression IS_CALLED JOIN IS_ENROLLED_ON gave us a relation representing the extension of the conjunctive predicate "Student *StudentId* is called *Name* and is enrolled on course *CourseId*". Now consider the *dis*junctive predicate "Student *StudentId* is called *Name* **or** is enrolled on course *CourseId*", the same as before except that the logical connective OR is used instead of AND.

Recall also that the truth table for AND (Figure 3.2) enabled us to determine which tuples, derived from those of IS_CALLED and IS_ENROLLED_ON, satisfy that conjunctive predicate and thus constitute the body of their join. Unfortunately, the truth table for OR (Figure 3.3) tells us that it's not so easy to discover all the tuples that satisfy the disjunctive predicate.

Let's think about the extension of "Student *StudentId* is called *Name* or is enrolled on course *CourseId*". Here is a true instantiation that we can determine by examination of the current values of IS_CALLED and IS_ENROLLED on:

Student S1 is called Anne or is enrolled on course C1.

So `TUPLE {StudentId SID('S1'), Name NAME('Anne'), CourseId CID('C1')}` clearly appears in the body of the corresponding relation. But the following instantiations are also true:

a) Student S1 is called Anne or is enrolled on course C3.
b) Student S1 is called Jane or is enrolled on course C1.
c) Student S1 is called Anne or is enrolled on course C4751.
d) Student S1 is called Xdfrtghyuz or is enrolled on course C1.

In case (a) the first disjunct, "Student S1 is called Anne", is true and the second is false. A similar remark applies to case (c), even though, as it happens, course C4751 doesn't even exist—the corresponding tuple appears in the relation because `CID('C4751')` does denote a value of type `CID` (and S1's name is Anne). Cases (b) and (d) connect true statements about enrolments with false ones about names. In (d), though we can perhaps be 100% certain that nobody has ever been called Xdfrtghyuz, I am assuming that `NAME('Xdfrtghyuz')` is a value of type `NAME` (it would be difficult to define the type in such a way as to exclude such possibilities).

You can see, then, that the extension of the *dis*junctive predicate contains an inordinately large number of instantiations compared with the extension of the much more restrictive *con*junctive predicate. In practical terms, the corresponding relation is too big, would take too much time to be computed, and in any case isn't very useful. For those reasons, Codd deliberately excluded a general relational counterpart of `OR` from his relational algebra. However, noting that *some* disjunctive predicates do not suffer from this problem, he did include a dyadic operator, `UNION`, to give relations representing just those special cases.

Codd noted that the relation for a predicate of the form *p* `OR` *q* is computable—the problem just described does not arise—if *p* and *q* have exactly the same set of parameters and the relations for *p* and *q* are computable. His `UNION` operator, therefore, requires its relation operands to be *of the same type* (i.e., to have the same heading).

Consider, then, the predicate

Student *StudentId* is called Devinder **OR** student *StudentId* is enrolled on course C1.

The two disjuncts both have just the one parameter, *StudentId*. The first disjunct is derived from the predicate for `IS_CALLED` by substituting a value for the *Name* parameter (thus binding it); the second is derived from the predicate for `IS_ENROLLED_ON` by similarly binding the *CourseId* parameter. Although the expression `IS_CALLED UNION IS_ENROLLED_ON` is illegal in **Tutorial D**, for the reason I have given, the expression given in Example 4.9 *is* legal.

Example 4.9: A legal invocation of UNION

```
( IS_CALLED WHERE Name = NAME ('Devinder') ) { StudentId }
UNION
( IS_ENROLLED_ON WHERE CourseId = CID ('C1') ) { StudentId }
```

The binding of *Name* is achieved in two steps. First we use restriction (`WHERE`) to restrict it to exactly one value by equals comparison; then we "project it away". We dispose of *CourseId* in similar fashion. The resulting relation is shown in Figure 4.11. You can verify it for yourself by checking whether each of the students S1, S2 and S4 is either called Devinder or enrolled on course C1.

StudentId
S1
S2
S4

Figure 4.11: Result of Example 4.9

Definition of `UNION`

Let *s* = *r1* **UNION** *r2*
where *r1* and *r2* are relations of the same type.

Then:

- The heading of *s* is the common heading of *r1* and *r2*.
- The body of *s* consists of each tuple that is *either* a tuple of *r1* *or* a tuple of *r2*.

Points to note:

- The cardinality of *s* is no less than the cardinality of the larger operand and no greater than the sum of the operand cardinalities. So, assuming the operands are computable, the result must be computable.
- Like `JOIN`, `UNION` is commutative. It is no mere coincidence that `OR` is also commutative.
- Also like `JOIN`, `UNION` is associative. It is no mere coincidence that `OR` is also associative. Because `UNION` is associative, we can define an *n*-adic version:

  ```
      UNION { r1, r2, … }
  ```
  where `UNION { r } = r.`
- Also like `JOIN`, `UNION` is idempotent: *r* `UNION` *r* = *r*.

- *Un*like `JOIN`, `UNION` does not have a single identity value. The normal set-theory union operator does have an identity value, namely, the empty set. But relations have headings and there is no single heading that can be the heading of a relation that, when unioned with any relation *r,* yields *r.* However, we *can* note that if *re* is the empty relation of the same type as *r,* then *r* `UNION` *re* = *r.* For that reason, **Tutorial D**, for what it's worth, does allow you to take the union of no relations at all, so long as you specify the heading of the desired result. For example:

  ```
  UNION { x CHAR, y INTEGER, z SID } { }
  ```

  If the key word `UNION` is replaced by `RELATION`, that would yield the same result of course.

Now you may well ask, what is the point of being able to take the union of no relations? Well, experience shows that it is a mistake in a computer language to legislate against support for empty operands when the operation in question has a mathematically respectable definition. Sometimes scripts in a desired language are generated for us by special-purpose software; legislating against something that is definable but apparently pointless sometimes turns out to be inconvenient for the developers of the generating software. I know of several cases in SQL, for example, where that kind of concern has arisen. In particular, standard SQL has no counterparts of `TABLE_DEE` and `TABLE_DUM`, nor does it allow one to express unions and joins with no operands.

There remains one logical operator that I have not given you a relational counterpart of yet: *negation*. We shall see that, as with `OR`, we have to restrict ourselves to certain special cases involving negation, and these cases are addressed by a dyadic operator that goes by the strange name *semidifference*.

4.10 Semidifference and NOT

Consider the negation of the predicate for `IS_CALLED`: "Student *StudentId* is **NOT** called *Name*." We run into the same problem as that one we encountered with disjunction, for its extension includes instantiations such as "Student S1 is not called Jane", "Student S97431 is not called Anne", "Student S1 is not called Xdfrtghyuz", and so on, *ad infinitum* ("almost"!). The term *unsafe* is sometimes used in connection with predicates such as these, where the corresponding relation is likely to be of such inordinately high cardinality as to be not computable in practice. It is no coincidence that both negation and disjunction are unsafe in general: recall that $p \lor q$ is equivalent to $\neg(\neg p \land \neg q)$.

So, when exactly *can* we "safely" use negation? Here is one example:

StudentId is called Devinder **AND** *StudentId* is **NOT** enrolled on C1.

Here we use `AND` to connect two predicates that have the same parameter—the same restriction as the one we needed for disjunction. In this case, though, the second is negated. In **Tutorial D** we can obtain the relation representing this predicate by using an operator named `MINUS`, as shown in Example 4.10. `MINUS` is a relational counterpart of set difference and is subject to exactly the same restriction as `UNION` concerning the types of its operands.

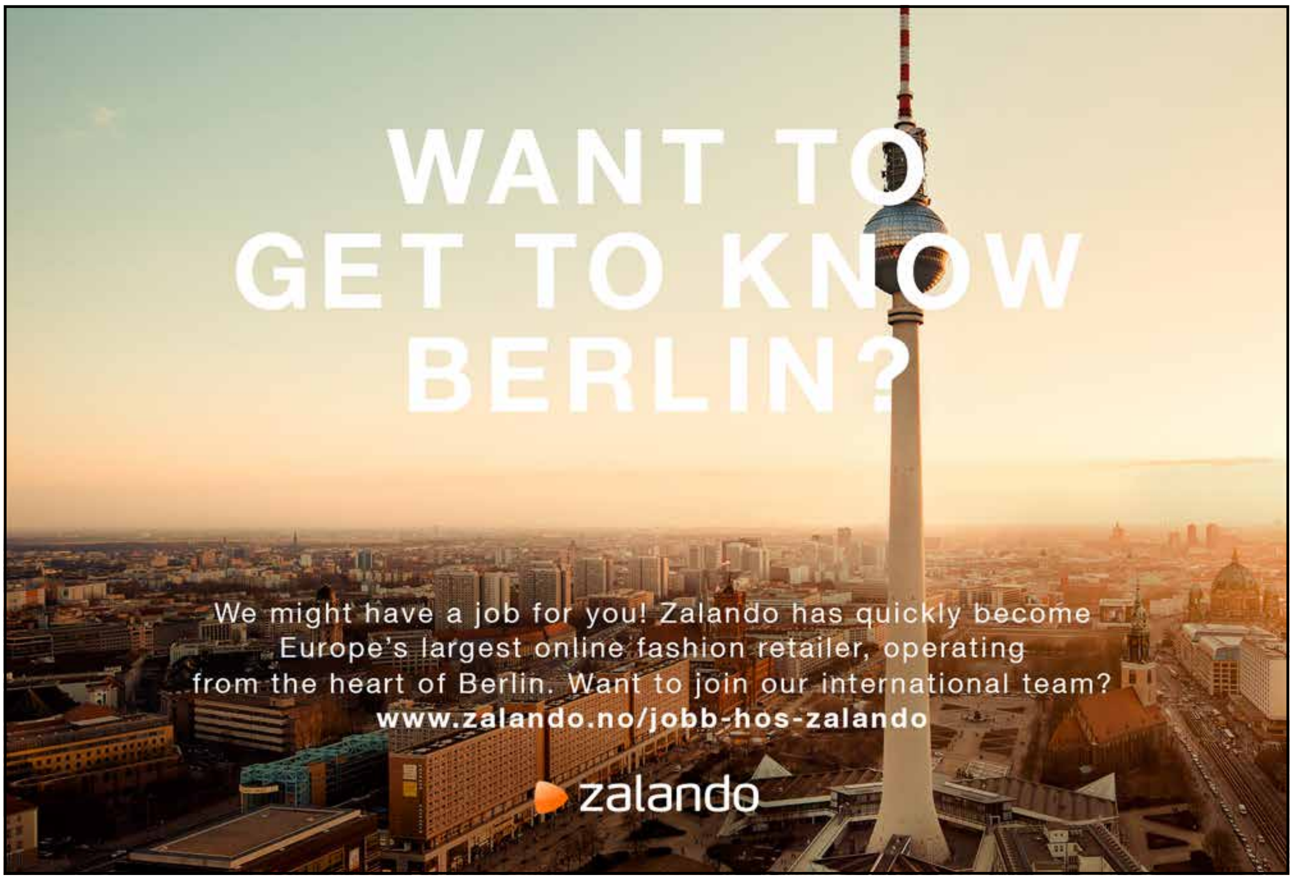
Example 4.10: A legal invocation of MINUS

```
( IS_CALLED WHERE Name = NAME ('Devinder') ) { StudentId }
MINUS
( IS_ENROLLED_ON WHERE CourseId = CID ('C1') ) { StudentId }
```

In general *r1* `MINUS` *r2* returns the relation whose heading is that of both operands and whose body consists of those tuples of *r1* that do not appear in the body of *r2*. `MINUS` was the operator proposed by Codd for dealing with negation, but it turned out that we can do better than that. In fact, we don't actually need that restriction on the operand types, if we base the operator on the concept of *matching* tuples, as we did with `JOIN`, instead of basing it on identical tuples in particular.

Consider the predicate

Student *StudentId* is called *Name* and *StudentId* is not enrolled on any course.


WANT TO
GET TO KNOW
BERLIN?
We might have a job for you! Zalando has quickly become
Europe's largest online fashion retailer, operating
from the heart of Berlin. Want to join our international team?
www.zalando.no/jobb-hos-zalando
zalando

The first conjunct is the predicate for `IS_CALLED`. The second is clearly derivable from the predicate for `IS_ENROLLED_ON` but we should write it a little more formally to see *how* it is derived:

> There does not exist a course *CourseId* such that student *StudentId* is enrolled on *CourseId*.

We can discover whether a given *StudentId* value satisfies this monadic predicate by looking to see if there is a tuple with that *StudentId* value in the current value of `IS_ENROLLED_ON`. That means that for each tuple in the current value of `IS_CALLED` we can test to see if the student referred to is enrolled on anything by looking to see if that tuple has a matching tuple in `IS_ENROLLED_ON`. Instead of taking the difference between certain projections to satisfy the requirement of `MINUS` for operands of the same type, we can take the *semidifference* between `IS_CALLED` and `IS_ENROLLED_ON` (in that order). **Tutorial D** uses the more intuitive name `NOT MATCHING` for this operator, as illustrated in Example 4.11.

Example 4.11: Semidifference (students not on any courses)

```
IS_CALLED NOT MATCHING IS_ENROLLED_ON
```

Figure 4.12 shows the result of Example 4.11.

StudentId	Name
S5	Boris

Figure 4.12: Result of Example 4.11

Definition of NOT MATCHING

Let *s* = *r1* **NOT MATCHING** *r2*
where *r1* and *r2* are relations such that *r1* JOIN *r2* is defined.

Then:

- The heading of *s* is the heading of *r1*.
- The body of *s* consists of each tuple of *r1* that has no matching tuple in *r2*.

Points to note:

- The body of the result is, as with restriction, a subset of that of the first operand. It follows that if *r1* is empty, then so is *s*.
- If *r1* and *r2* have no common attributes, then *s* is equal to *r1* in the case where *r2* is empty and is otherwise empty (note that tuples having no common attributes are matching tuples under the definition of that term given in Chapter 4, Section 4.4 **JOIN and AND**).
- Where *r1* `MINUS` *r2* is defined, it is equivalent to *r1* `NOT MATCHING` *r2*. So we can clearly dispense with `MINUS` without sacrificing completeness. As an exercise, you might like to check whether an algebra that, like Codd's, includes `MINUS` instead of `NOT MATCHING` also does not thereby sacrifice completeness.

It's a natural question to ask whether `MATCHING` is also supported. In fact it is, but because its inclusion in the language is not essential for relational completeness I defer discussion of it to the next chapter. In a nutshell, *r1* `MATCHING` *r2* yields the relation whose body consists of just those tuples of *r1* that have at least one matching tuple in *r2*.

4.11 Concluding Remarks

I have described the following relational operators:

- `JOIN`
- `RENAME`
- projection
- restriction (`WHERE`)
- `EXTEND`
- `UNION`
- semidifference (`NOT MATCHING`)

I have claimed that these meet the criterion for completeness of a relational algebra for computational purposes. I justify this claim by showing that they include relational counterparts of logical `AND`, `OR`, `NOT`, and existential quantification, with agreed limitations on `OR` and `NOT` in the interests of "safety". I have left slightly open the question of whether the chosen set is *necessary* for completeness, as well as being sufficient for it. Let me address that point now.

Why do we have three different counterparts of `AND`? Well, it is true that if our purpose were mathematical only we could use `JOIN` for restrictions and extensions too, but that would entail the use of operand relations representing operators like "<" and "+", and these tend to be as inordinately large as those that prohibit us from having unrestricted relational counterparts for disjunction and negation. For consider, the relation for "<" of integers includes a tuple for every pair <*a,b*> of integers such that *a*<*b* is true, and the relation for plus includes a tuple for every triple <*a,b,c*> such that *a*+*b*=*c* is true. Restriction and extension allow us to use invocations of operators in open expressions.

That said, there is actually one operator in my list that could be dispensed with without sacrificing completeness—one whose definition I could have given in terms of others in my list. I leave it as an exercise for the reader to discover which one that is and write a definition for it using the other operators. In any case, the inclusion of this superfluous operator—and we do have good reason to include it—reminds us all that we don't always want an algebra to be pared to the theoretical minimum in terms of its number of operators. In propositional logic, for example, the psychological reasons for including both `AND` and `OR`, even though either can be defined in terms of the other along with `NOT`, are, most would agree, overwhelming.

I have laid the foundation. In the next chapter I build on that foundation.

EXERCISES

1. Recall that *r1* `TIMES` *r2* requires *r1* and *r2* to have no common attributes, in which case it is equivalent to *r1* `JOIN` *r2*. Why would it be a bad idea to *require* `TIMES` to be used in place of `JOIN` in such cases?

2. Given the following relvars:
```
VAR Cust BASE RELATION {C# CHAR, Discount RATIONAL} KEY {C#};
VAR Orders BASE RELATION {O# CHAR, C# CHAR, Date DATE} KEY {O#};
VAR OrderItem BASE RELATION {O# CHAR, P# CHAR, Qty INTEGER }
                     KEY {O#, P#};
VAR Product BASE RELATION {P# CHAR, Unit_price RATIONAL}
                     KEY {P#};
```
   The price of an order item can be calculated by the formula:
```
  CAST_AS_RATIONAL(Qty)*Unit_price*(1.0-(Discount/100.0))
```
   Write down a relation expression to yield a relation with attributes O#, P#, and PRICE, giving the price of each order item.

3. Given:
```
VAR Exam_Marks BASE RELATION { StudentId SID,
                               CourseId CID,
                               Mark INTEGER}
                          KEY { StudentId, CourseId };
```
   Write down a relational expression to give, for each pair of students sitting the same exam, the absolute value of the difference between their marks. Assume you can write `ABS`(*x*) to obtain the absolute value of *x*.

4. State the value of
   a) *r* `NOT MATCHING TABLE_DEE`
   b) *r* `NOT MATCHING TABLE_DUM`
   c) *r* `NOT MATCHING` *r*
   d) (*r* `NOT MATCHING` *r*) `NOT MATCHING` *r*
   e) *r* `NOT MATCHING` (*r* `NOT MATCHING` *r*)
   Is `NOT MATCHING` associative? Is it commutative?

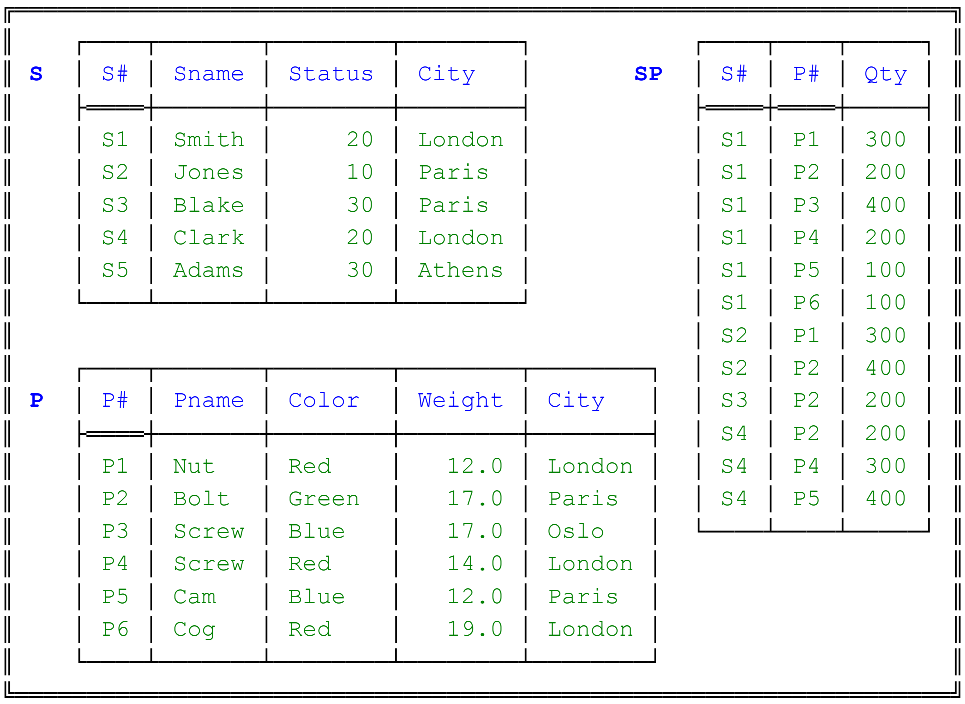
5. (Repeated from the body of the chapter) Which operator, in the list given in Section 4.11, **Concluding Remarks**, can be dispensed with without sacrificing relational completeness? How can it be defined in terms of the other operators?

6. (Repeated from the body of the chapter) Investigate the completeness of an algebra that includes `MINUS` in place of `NOT MATCHING` by attempting to define `NOT MATCHING` in terms of `MINUS` and the other operators.

7. The chapter briefly mentions the operator `MATCHING` but defers its detailed description to Chapter 5. Before you read that chapter, define *r1* `MATCHING` *r2* in terms of the operators described in Chapter 4.

Working with a Database in *Rel*

1. Start up *Rel*.
2. Figure 4.13 shows the supplier-and-parts database from Chris Date's *Introduction to Database Systems (8th edition),* as shown on the inside back cover of that book (except that the attribute names there are in upper case).



S

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

S#	P#	Qty
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

P

P#	Pname	Color	Weight	City
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

Figure 4.13: The suppliers-and-parts database

Execute a **Tutorial D** `VAR` statement for each of `S, P` and `SP`. Use `INTEGER` as the declared type for `STATUS` and `QTY, RATIONAL` for `WEIGHT,` and `CHAR` for all the other attributes. Feel free to use lower case or mixed case to suit your own taste for attribute and relvar names, but do not otherwise change any of the given names.

Tutorial D requires at least one key constraint to be specified for each relvar. One key for each for `S`, `P` and `SP` is shown by underlining the relevant attribute names in the table. No other key constraints are needed.

"Populate" (as they say) each relvar with the values shown in Date's tables. There are several ways of achieving this. Choose whichever you prefer from the following:

a) Include an `INIT ( ... )` specification in the `VAR` statement, after the heading and before the `KEY` specification. Inside the parens, write a `RELATION { ... }` expression, using a `TUPLE` expression for each required tuple, as in the enrolment literal used in the *Rel* exercises for Chapter 2.

b) Execute the `VAR` statement without an `INIT ( ... )` specification. The implied initial value is the empty relation of the appropriate type. You can see this by asking *Rel* for the current value of the relvar. For example, to get the current value of `S`, just type `S` into the lower pane and click `Run (F5)`.
Now use an assignment statement of the form
`varname := relation-expression`
to populate the relvar. Check that *Rel* has indeed assigned the correct value to it.

c) Use *Rel* `INSERT` statements to populate the relvar piecemeal, perhaps one tuple at a time. Having typed in the first `INSERT` statement. Here is the general form of an `INSERT` statement to insert a single tuple:

```
        INSERT varname RELATION { TUPLE { ... } } ;
```

Note that the source operand is still a relation, not just a tuple, hence the need to enclose the `TUPLE` expression inside `RELATION { }`.

3. Informally, we refer to `S` as suppliers, `P` as parts and `SP` as shipments. Predicates for these relvars are:
`S:` Supplier *S#* is named *Sname*, has status *Status* and is located in city *City*.
`P:` Part *P#* is named *Pname*, is coloured *Color*, weighs *Weight* and is located in city *City*.
`SP:` Supplier *S#* ships part *P#* in quantities of *Qty*.
What, then, is the predicate for the expression `S JOIN SP JOIN P`? What do you expect to be the result of that expression? What is its degree? Does *Rel* give the result you expected? Explain what you see.
4. Attempt to insert a tuple into `SP` with supplier number S1, part number P1 and quantity 100. Explain the result of your attempt.
5. Get *Rel* to evaluate each of the following expressions. For each one, write down the corresponding predicate and also give an informal interpretation of the query in the style of those given in Exercise 6 below.
   a) `SP WHERE P# = 'P2'`
   b) `S { ALL BUT Status }`
   c) `SP { S#, Qty }`
   d) `P NOT MATCHING ( SP WHERE S# = 'S2' )`
   e) `S MATCHING ( SP WHERE P# = 'P2' )`

f) `S { City } UNION P { City }`

g) `S { City } MINUS P { City }`

h)
```
( ( S RENAME { City AS SC } ) { SC } ) JOIN
( ( P RENAME { City AS PC } ) { PC } )
```

6. Write **Tutorial D** expressions for the following queries and get *Rel* to evaluate them:
   a) Get all shipments.
   b) Get supplier numbers for suppliers who supply part P1.
   c) Get suppliers with status in the range 15 to 25 inclusive.
   d) Get part numbers for parts supplied by a supplier in Paris.
   e) Get part numbers for parts not supplied by any supplier in Paris.
   f) Get city names for cities in which at least two suppliers are located.
   g) Get all pairs of part numbers such that some supplier supplies both of the indicated parts.
   h) Get supplier numbers for suppliers with a status lower than that of supplier S1.
   i) Get supplier-number/part-number pairs such that the indicated supplier does not supply the indicated part.

5 Building on The Foundation

5.1 Introduction

The relational operators described in Chapter 4 constitute a relationally complete set; but for practical purposes we need more. This chapter describes additional operators, defined for **Tutorial D**, which have been suggested as desirable by various people over the years. The additional relational operators are all defined in terms of those defined in Chapter 4.

The relvars `IS_CALLED` and `IS_ENROLLED_ON` have been sufficient for illustrative purposes so far but now we need to extend our example database slightly, as shown in Figure 5.1.

COURSE

CourseId	Title
C1	Database
C2	HCI
C3	Op systems
C4	Programming

EXAM_MARK

StudentId	CourseId	Mark
S1	C1	85
S1	C2	49
S1	C3	85
S2	C1	49
S3	C3	66
S4	C1	93

Figure 5.1: Current values of relvars COURSE and EXAM_MARK

(We assume that the relvar IS_ENROLLED_ON now includes a tuple for student S1 on course C3.)

The predicate for `COURSE` is "Course *CourseId* is entitled *Title*." The predicate for `EXAM_MARK` is "Student *StudentId* sat the exam for course *CourseId* and scored *Mark* marks for that exam." The **Tutorial D** definitions for these relvars are

```
VAR COURSE BASE RELATION { CourseId CID, Title CHAR }
                    KEY { CourseId };
VAR EXAM_MARK BASE RELATION { StudentId SID, CourseId CID,
                              Mark INTEGER }
                    KEY { StudentId, CourseId };
```

We start with a couple of simple operators, based on ones already described in Chapter 4, that provide convenient *shorthands* (so called because what they express can be expressed more laboriously, and perhaps less clearly, without the use of the additional relational operators).

5.2 Semijoin and Composition

Consider the predicate, "At least one student sat the exam for Course *CourseId*, entitled *Title*"—more precisely: "There exist a student *StudentId* and a mark *Mark* such that *StudentId* sat the exam and scored *Mark* marks for course *CourseId* and *CourseId* is entitled *Title*."

The relation currently representing this predicate can be derived from the join of COURSE and EXAM_MARK by "projecting away" the attributes corresponding to those quantified parameters, *StudentId* and *Mark*:

```
( COURSE JOIN EXAM_MARK ) { ALL BUT StudentId, Mark }
```

or, equivalently,

```
COURSE JOIN ( EXAM_MARK { ALL BUT StudentId, Mark } )
```

In either case the JOIN is indicated for us by the "and" in the expanded version of the predicate and the projection is indicated by the quantification. However, looking at the short form of the predicate we may more intuitively think of its relation as consisting of just those tuples of COURSE that have at least one matching tuple in EXAM_MARK. Now we may recall from Chapter 4 that we can find all the tuples of COURSE that do *not* have a matching tuple in EXAM_MARK by using semidifference:

```
COURSE NOT MATCHING EXAM_MARK
```

and I indicated briefly that **Tutorial D** also allows you to omit the word NOT, with the obvious effect:

```
COURSE MATCHING EXAM_MARK
```

MATCHING, without the NOT, is **Tutorial D**'s operator name for *semijoin,* so called because a semijoin can be perceived, very loosely, as being "half a join". We join two relations but in the result retain only the attributes of the first operand (by excluding the non-common attributes of the second). That result is shown in Figure 5.2. It contains every tuple of COURSE apart from the tuple for course C4, whose exam no student sat.

CourseId	Title
C1	Database
C2	HCI
C3	Op systems

Figure 5.2: COURSE MATCHING EXAM_MARK

Now, perhaps, you can see why somebody once chose "semidifference" for the operator of that name, even though it can hardly be characterized as "half a difference": the name "semijoin" had already entered the jargon. (No, there isn't a semiunion!)

Definition of `MATCHING`

r1 **MATCHING** *r2,* where *r1* and *r2* are relations such that *r1* JOIN *r2* is defined, is equivalent to
(*r1* **JOIN** *r2*) { *r1-attrs* }
where *r1-attrs* is a commalist containing all and only the attribute names of *r1*.

Points to note (compare with those given for `NOT MATCHING` in Chapter 4):

- Recall that `JOIN` is not defined for all pairs of relations. If *r1* and *r2* have attributes of the same name but different types, then *r1* `JOIN` *r2* is not defined. A similar proviso applies to `MATCHING` and several other dyadic operators.
- The body of the result is, as with restriction, a subset of that of the first operand. It follows that if *r1* is empty, then so is the result.
- If *r1* and *r2* have no common attributes, then the result is empty in the case where *r2* is empty and is otherwise equal to *r1* (recall that tuples having no common attributes are considered to be matching tuples).
- As the definition shows, semijoin is not needed as a primitive operator. As explained in Chapter 4, we could have chosen difference in place of semidifference as our primitive operator to support logical negation, but we preferred semidifference for its more general availability. Had we chosen difference instead, then our descriptions of `NOT MATCHING` and `MATCHING` could have appeared, neatly, side by side in the present chapter.

There is another operator, advocated by some people as being useful enough to warrant its inclusion, that is based, like semijoin, on `JOIN` and projection. It is called *composition*.

Consider the predicate "Student *StudentId* scored *Mark* marks in the exam for a course entitled *Title*." The corresponding relation must have attributes `StudentId`, `Mark`, and `Title`. The first two would clearly be derived from `EXAM_MARK`, the third from `COURSE`.

Notice the indefinite article in that predicate: "*a* course", not "*the* course". Here we can replace the word "a" by "some" without changing our meaning at all, indicating that existential quantification is lurking under the covers, so to speak, in our informal predicate. We can bring that quantification out into the open, as I did with the example I used for semijoin: "There exists a course *CourseId* such that *CourseId* is entitled *Title* and student *StudentId* sat the exam for *CourseId*, scoring *Mark* marks."

The relation representing this predicate can be derived from the join of COURSE and EXAM_MARK by "projecting away" *CourseId* (which happens to be the only common attribute):

```
    ( COURSE JOIN EXAM_MARK ) { ALL BUT CourseId }
```

Composition gives us a shorthand that saves us from having to write that projection when its purpose is to exclude all and only the common attributes of the two operand relations. In **Tutorial D**, therefore, we can achieve the same effect more conveniently by

```
    COURSE COMPOSE EXAM_MARK
```

The result is shown in Figure 5.3.

Title	StudentId	Mark
Database	S1	85
HCI	S1	49
Op systems	S1	85
Database	S2	49
Op systems	S3	66
Database	S4	93

Figure 5.3: COURSE COMPOSE EXAM_MARK

Definition of COMPOSE

r1 **COMPOSE** *r2,* where *r1* and *r2* are relations such that *r1* `JOIN` *r2* is defined, is equivalent to

(*r1* `JOIN` *r2*) { `ALL BUT` *common-attrs* }

where *common-attrs* is a commalist containing all and only the names of the attributes common to *r1* and *r2.*

Points to note:

- *r1* `COMPOSE` *r2* is clearly equivalent to *r1* `JOIN` *r2* in the case where *r1* and *r2* have no common attributes.
- The case where *r1* and *r2* have identical headings is worth looking at. I invite the reader to study that case and find out what happens.
- Like `JOIN`, `COMPOSE` is commutative—that's clear, but is it also associative? Again, I leave that question as an exercise for the reader—can you find an example where (*r1* `COMPOSE` *r2*) `COMPOSE` *r3* does not yield the same result as *r1* `COMPOSE` (*r2* `COMPOSE` *r3*)?

In case you are wondering if `COMPOSE` really is useful enough to be worth including in a computer language, and therefore to be worthy of inclusion in textbooks like this one, an important part of the motivation for its inclusion in **Tutorial D** was a desire to illustrate the *extensibility* of a well-designed language. Adding new operators increases a language's complexity, to be sure, but that added complexity can be compensated for if the new operators are not only useful but can be easily defined and taught in terms of what the user already knows.

The term *composition* as used here comes from mathematics, where it is used of *functions*. The explanation is not important for our purposes but I give it here for those that may be interested.

A function is a special kind of binary relation, usually described as a *mapping* that connects each element of a set, known as the *domain* of the function, to exactly one element of another set (possibly the same set), known as the *range* of the function. Our relvar `COURSE` is in fact a function—or rather, its value at any point in time is a function—mapping each element of a certain set of course identifiers to exactly one element of a certain set of titles. Similarly, `EXAM_MARK` maps each element of a certain set of <student identifier, course identifier> pairs to exactly one element of a certain set of marks out of 100. `EXAM_MARK` also maps <student identifier, mark> pairs to course identifiers, but that mapping is not a function because it is possible for the same <student identifier, mark> pair to be connected to several distinct course identifiers, the pair <S1, 85> being a case in point.

In mathematics, the composition of two functions *f* and *g* is defined in the case where the range of *g* is the domain of *f*. Thus, if *g*(*x*) denotes the element of *g*'s range to which its domain element *x* is connected, then *f*(*g*(*x*)) denotes the element of *f*'s range to which the element *g*(*x*) (of *f*'s domain) maps. The composition of *f* and *g* is thus a function whose domain is the domain of *g* and whose range is a subset of the range of *f*. The set that is the range of *g* and the domain of *f* in a sense "disappears" in the process of deriving the composition from the two participating functions.

Well, just as a function is a special kind of relation, we can regard composition of functions as a special case of composition of relations. In `COURSE COMPOSE EXAM_MARK` we map <student identifier, mark> pairs to course titles via their mapping to course identifiers, losing those course identifiers in the process. (But note that the mapping here is many-to-many, not, as in functions, many-to-one.)

Notice, by the way, that if two courses happened to have the same title, and a student who sat the exam for both of those courses scored the same mark in each case, then that fact would be represented by just one tuple in the result of `COURSE COMPOSE EXAM_MARK`. We cannot safely deduce from that result the total number of exams taken by students, unless course titles are unique as well as course identifiers—unless, that is, `COURSE` represents a function mapping titles to identifiers as well as one mapping identifiers to functions. It seems that care needs to be taken over cases of relation composition that do not in fact represent function composition—one might easily misinterpret the result.

Vi vokser i Norge og har virksomhet helt frem til 2050
Er du interessert i sommerjobb eller fast stilling?
bp
Se informasjon om sommerjobber på www.bp.no

Next, we look at a group of operators that, when invoked, operate on relations but do not return relations: *aggregate operators*. With these added to our "tool box" we can then proceed to define further useful shorthands.

5.3 Aggregate Operators

An aggregate operator is one defined to operate on a relation and return a value obtained by aggregation over all the tuples of the operand. For example, simply to count the tuples in the body of `EXAM_MARK` (i.e., obtain its cardinality) we can invoke the aggregate operator `COUNT`, as shown in Example 5.1.

Example 5.1: Counting the tuples in EXAM_MARK

```
COUNT ( EXAM_MARK )
```

According to Figure 5.1, the result of `COUNT ( EXAM_MARK )` is 6. The argument to an invocation of `COUNT` is a relation and so in **Tutorial D** can be denoted by any legal relational expression. Example 5.2 gives the number of students who have scored more than 50 in at least one exam.

Example 5.2: Using relational operators with COUNT

```
COUNT ( ( EXAM_MARK WHERE Mark > 50 ) {StudentId} )
```

Note the projection over `StudentId`. Without that, the expression would yield the (possibly higher) number of students' exam scripts scoring more than 50.

Now take a look at Example 5.3.

Example 5.3: Aggregate operator SUM

```
SUM ( EXAM_MARK WHERE StudentId = SID ( 'S1' ), Mark )
```

Note the second operand, `Mark`, being the name of an attribute of the relation denoted by the first operand. Each tuple of the first operand provides a value for this attribute and the result of the invocation is the sum of those values. The result in this example is 219, the sum of the scores obtained by student S1. Note that both appearances of the score 85 in S1's marks are counted.

Example 5.4: MAX and MIN

```
MAX ( EXAM_MARK WHERE StudentId = SID ( 'S1' ), Mark )
MIN ( EXAM_MARK WHERE StudentId = SID ( 'S1' ), Mark )
```

Two more aggregate operators are illustrated in Example 5.4. `MAX` returns the highest value found for the specified attribute in the given relation and `MIN` returns the lowest.

Points to note:

- Some aggregations can be thought of in terms of repeated invocation of some dyadic operator, which I shall call the *basis operator*. In the case of `SUM`, for example, the basis operator is addition. Because addition is commutative and associative, we could define an *n*-adic form of the operator, just as we did in Chapter 4 for operators such as `JOIN` and `UNION`. If we call this operator `ADD`, then we would have, for example, `ADD(1,4,1,5) = ((1+4)+1)+5`. But those operands, 1, 4, 1, and 5, can be given in any order (thanks, in this case, to the commutativity and associativity of `+`), and that lack of any significance to the ordering is what allows us to define aggregate operators for relations. The lack of an ordering to the tuples of a relation militates against defining aggregate operators whose results vary according to the order in which the operands are presented. Consider string concatenation, for example. We can concatenate any number of strings together to form a single string, but the result depends on the order in which the input strings are presented.
- The basis operators for `MAX` and `MIN` might reasonably be called `HIGHER` and `LOWER`, respectively, where `HIGHER(`*x,y*`)` returns *x* unless *y>x,* in which case it returns *y,* and `LOWER(`*x,y*`)` returns *x* unless *y<x,* in which case it returns *y.* You can confirm for yourself that `HIGHER` and `LOWER` are commutative and associative.
- If the relation operand is empty, then the result of aggregation can be defined only if the basis operator has an *identity value*, defined thus: if a value *i* exists such that whenever *i* is one of the operands of a dyadic operator the result of invoking that operator is the other operand, then *i* is said to be an identity value under that operator. In the case of `SUM`, the basis operator is addition, whose identity value is zero. In the cases of `MAX` and `MIN`, the type of the result is the type of the attribute given as the second operand. The identity value of the basis operator depends on that type. If the type has a defined least value, *min*, such that *min>v* is `FALSE` for all values *v* of that type, then *min* is the identity under `HIGHER`. If a least value is not defined, then there is no identity value under `HIGHER`, and `MAX` of the empty relation is undefined for attributes of that type. Similarly, `MIN(`*r,a*`)` is defined only when a greatest value is defined for the type of attribute *a*.
- The examples shown use a simple attribute name as the second operand, and Version 1 of **Tutorial D** in fact requires that operand to be a simple attribute name. In general, however, the second operand in invocations of `SUM`, `MAX`, and `MIN` should be allowed to be any expression of an appropriate type (obviously a numeric type in the case of `SUM`). Version 2 of **Tutorial D** does indeed allow this.

- The simple attribute names used in my examples are cases of open expressions, as defined in Chapter 4, Section 4.7. As in other places where open expressions are permitted, closed expressions are also permitted—allowing us to sagely observe, for example, that `SUM(`*r*`,1)` is equivalent to `COUNT(`*r*`)`.

Several other aggregate operators are defined in **Tutorial D**. Here are some that we can now deal with summarily:

> **AVG** (*r*, *x*) is equivalent to `SUM` (*r*, *x*) / `COUNT` (*r*) and is therefore undefined in the case where *r* is empty. As an exercise, the reader might like to consider whether there can be a basis operator for `AVG`.

AND (*r*, *c*) and **OR** (*r*, *c*), where *c* (a condition) is of type `BOOLEAN`, are named after their basis operators—recall that logical `AND` and `OR` are commutative and associative, with identity values `TRUE` and `FALSE`, respectively. Thus, aggregate `AND` returns `TRUE` if and only if *c* evaluates to `TRUE` for every tuple of *r*; and aggregate `OR` returns `TRUE` if and only if *c* evaluates to `TRUE` for some tuple of *r*. In some languages the names `ALL` and `SOME` (or `ANY`) are used in place of `AND` and `OR`. (Indeed, *Rel* allows `ALL` and `ANY` to be used as synonyms for `AND` and `OR`.) Some people find it counterintuitive that aggregate `AND` on an empty relation returns `TRUE` but this is of course a logical necessity: to say that *c* is `TRUE` for every tuple in *r* is the same as saying there does not exist a tuple in *r* for which *c* is `FALSE`.

Now, suppose we want to find out how many students sat each exam. Do we have to go to the lengths illustrated in Example 5.5? I have used *Rel*'s explicit `OUTPUT` statements in that example to emphasise that it involves four distinct queries, one for each course. That would be very tiresome if we had a very large number of courses to consider, impossible if we didn't even know all the course ids.

Example 5.5: Number of students who sat each exam

```
OUTPUT COUNT ( EXAM_MARK WHERE CourseId = CID ( 'C1' ) );
OUTPUT COUNT ( EXAM_MARK WHERE CourseId = CID ( 'C2' ) );
OUTPUT COUNT ( EXAM_MARK WHERE CourseId = CID ( 'C3' ) );
OUTPUT COUNT ( EXAM_MARK WHERE CourseId = CID ( 'C4' ) );
```

Shouldn't we be able to use just a single query to obtain the desired result, which is shown in Figure 5.4? After all, I have claimed that the operators described in Chapter 4 make **Tutorial D** relationally complete, so we should, if we support counting at all, be able to obtain the relation representing the predicate, "*n* students sat the exam for course *CourseId*".

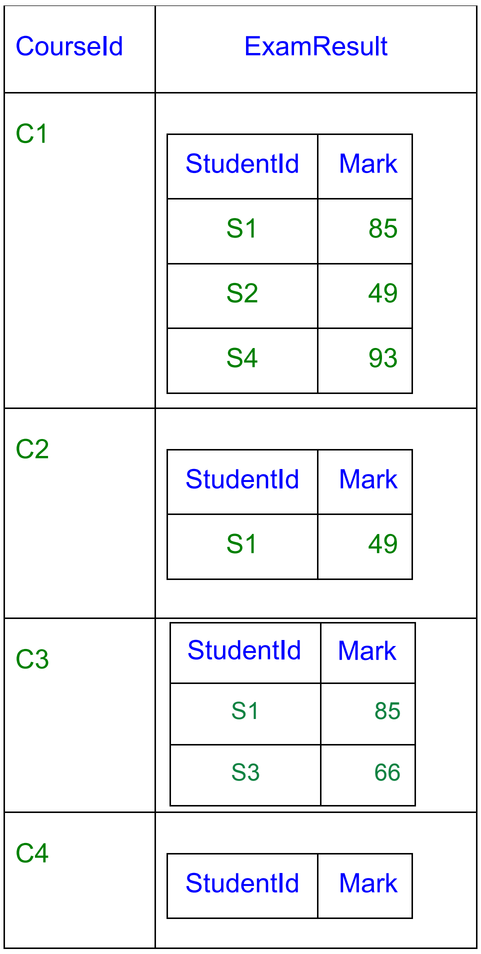
CourseId	n
C1	3
C2	1
C3	2
C4	0

Figure 5.4: How many sat each exam

The answer is that we can indeed obtain that relation using a single query and the next section starts to show you the way.

5.4 Relations within a Relation

Have a look at Figure 5.5. The figure itself is a two-column table, with a two-column table appearing in every cell of its second column {"second" because columns of tables do necessarily appear in some order, unlike attributes of relations}. The table depicts a relation—let's call it `C_ER`—whose attribute named `ExamResult` is of a certain relation type, namely, `RELATION { StudentId SID, Mark INTEGER }`.



CourseId	ExamResult
C1	StudentId | Mark S1 | 85 S2 | 49 S4 | 93
C2	StudentId | Mark S1 | 49
C3	StudentId | Mark S1 | 85 S3 | 66
C4	StudentId | Mark

Figure 5.5: Relations within a relation

Perhaps you have already noticed that the information represented by the table in Figure 5.5 is exactly the same as that represented by the tables in Figure 5.1, but in a different form. That being the case, we should be able to use relational operators to derive the relation `C_ER` from the current values of `COURSE` and `EXAM_MARK`. In fact it can be done using operators I have already described, as shown in Example 5.6.

Example 5.6: Obtaining C_ER from COURSE and EXAM_MARK

```
EXTEND COURSE{CourseId} :
{ ExamResult := RELATION { TUPLE { CourseId CourseId } }
                       COMPOSE EXAM_MARK }
```

Explanation 5.6

- The open expression on which the attribute `ExamResult` is defined denotes a relation, so the declared type of `ExamResult` is a relation type.
- **`TUPLE { CourseId CourseId }`** is an open expression, evaluated for each tuple in turn of the relation denoted by `COURSE{CourseId}`. It denotes the tuple of degree 1 the value of whose only attribute, `CourseId`, is the value of the attribute of that name in the current tuple of `COURSE{CourseId}`. In case you are puzzled by the consecutive appearances of `CourseId`, recall that each component of a `TUPLE` expression is an attribute name followed by an expression denoting the value for that attribute.
- **`RELATION`** `{ TUPLE { CourseId CourseId } }` denotes the relation whose body consists of just that tuple.
- `RELATION { TUPLE { CourseId CourseId } }` **`COMPOSE EXAM_MARK`** denotes the relation whose body consists of projections over `StudentId` and `Mark` of the tuples of `EXAM_MARK` that match `TUPLE { CourseId CourseId }`. Recall that *r1* `COMPOSE` *r2* denotes the join of *r1* and *r2* projected over all but the common attributes—here there is a single common attribute, `CourseId`.
- The value for the attribute `ExamResult` in the tuple for a given `CourseId` value *ci*, considering the method by which it is derived, is referred to as the *image relation* within the relation `EXAM_MARK` for the tuple `TUPLE { CourseId` *ci* `}`. In general, if *t* is a tuple and *r* is a relation, the image relation (or just image) of *t* in *r* is given by

  ```
  (RELATION { t }){ca} COMPOSE r
  ```

  where *ca* is the common attributes—attributes of *t* that are also attributes of *r*.

The values for attribute `ExamResult` in `C_ER` have sometimes been referred to informally as *nested relations,* being "relations within a relation", so to speak. Because those attribute values are relations, we can use aggregate operators on them. With that hint you should now be able to see how to obtain the result shown in Figure 5.4—how many students sat each exam.

5.5 Using Aggregate Operators with Nested Relations

Example 5.7 uses the aggregate operator COUNT on the relation values for attribute ExamResult to obtain the number of students who sat each exam.

Example 5.7: How many students sat each exam

```
WITH (
   EXTEND COURSE{CourseId} :
   { ExamResult := RELATION { TUPLE { CourseId CourseId } }
                       COMPOSE EXAM_MARK }
   AS C_ER ) :
EXTEND C_ER : { n := COUNT ( ExamResult ) }
{ ALL BUT ExamResult }
```

To avoid an indigestible surfeit of parentheses we use WITH to define the name C_ER I have been using in the text. Then we use extension to obtain the student counts as values for attribute n, followed by projection to eliminate the nested relations we no longer need. Similarly, we could obtain the total marks for each exam by including an extend addition such as TotalMarks := SUM(ExamResult, Mark), but if we need the average mark for each exam we will have to avoid zero-divides by excluding those which no students sat. We can do that by homing in on just those CourseId values that appear in EXAM_MARK, as shown in Example 5.8, the significant difference from Example 5.7 being shown in bold.

Example 5.8: Average mark per exam

```
WITH
   ( C_ER2 :=
         EXTEND EXAM_MARK{CourseId} :
        { ExamResult := RELATION { TUPLE { CourseId CourseId } }
                                  COMPOSE EXAM_MARK } ) :
EXTEND C_ER2 :
          { AvgMark := AVG(ExamResult, Mark) }
{ ALL BUT ExamResult }
```

Now, the expressions in Examples 5.7 and 5.8 are somewhat cumbersome and have much in common. They both use composition within extension to operate in a join-like fashion on two relations to produce a relation with a relation-valued attribute, `ExamResult`; and they both use extension on that relation to obtain results of aggregation on the nested relations, followed by projection to eliminate those nested relations. This commonality is captured in the definition of an operator named `SUMMARIZE`, providing useful shorthands for expressions such as the ones in Examples 5.7 and 5.8.

5.6 SUMMARIZE

Example 5.9 uses `SUMMARIZE` to give the same result as Example 5.7.

Example 5.9: How many students sat each exam, using SUMMARIZE

```
SUMMARIZE EXAM_MARK PER ( COURSE { CourseId } ) :
                        { n := COUNT ( ) }
```

Explanation 5.9

- **`EXAM_MARK PER ( COURSE { CourseId } )`** effectively gives `C_ER` but without specifying the attribute name for the attribute whose values are the nested relations. The missing attribute name doesn't matter because this intermediate result is never seen by a user.
- **`COUNT ( )`** represents an invocation of the aggregate operator `COUNT`, where the relation operand is implied—for each tuple of the `PER` relation (viz., `Course` projected over `CourseId`), it is the relation obtained as a value for `ExamResult` in Example 5.7.
- **`n :=`** `COUNT ( )` is a *summarize addition*, similar to an extend addition except that the expression on the right-hand side of := must be one that entails aggregation, thus obtaining a single attribute value from the implied relation operand. **Tutorial D** allows a
- commalist of summarize additions to be specified.
- As usual, that commalist is permitted to be empty. If `{ }` had been specified instead of `{ n := COUNT ( ) }`, then the expression would have been equivalent to just `COURSE { CourseId }`.
- **`SUMMARIZE … : { n :=`**`COUNT ( )` **`}`** extends the result of `EXAM_MARK PER ( COURSE { CourseId } )` as in Example 5.7. The fact that the nested relations do not appear as values for an attribute obviates the need for projection to exclude that attribute.
- In this example `EXAM_MARK` is the *SUMMARIZE operand*; `COURSE { CourseId }` is the `PER` *relation*.

The expression `COUNT ( )` (a kind of open expression) is called a *summary*. If we had wanted the total mark for each exam we would use the summary `SUM ( mark )`—again, the relation operand of the aggregate operator of the same name is omitted. Summaries are permitted only within invocations of `SUMMARIZE`, as shown.

In **Tutorial D** `SUMMARIZE` comes in two varieties. The variety used in Example 5.9 is called `SUMMARIZE PER`.

Definition of `SUMMARIZE PER`

SUMMARIZE *r1* **PER (** *r2* **) : {** *a1* **:=** *sum1*, …, *an* := *sumn* **}**, where:

- *r1* and *r2* are relations such that *r1* JOIN *r2* is defined,
- *a1*, …, *an* are attribute names not used in the heading of *r1*, and
- *sum1*, …, *sumn* are summaries

is equivalent to

`EXTEND` *r2* : { *a1* := *xsum1*, …, *an* := *xsumn* }

where each of *xsum1*, …, *xsumn* is an aggregate operator invocation such that:

- its relation operand is given by
  (`RELATION` { `TUPLE` { *b1b1*, …, *bm bm* } } `COMPOSE` *r1* },
  where *b1*, …, *bm* are the attributes of *r2*
- the aggregate operator and the remaining operands, if any, are as specified in the corresponding summaries *sum1*, …, *sumn*.

We can also use `SUMMARIZE PER` to obtain the result of Example 5.8, the average mark for each exam, as shown in Example 5.10. As in Example 5.8, we must restrict ourselves to the `CourseId` values appearing in `EXAM_MARK`.

Example 5.10: Average mark for each exam, using SUMMARIZE … PER …

```
SUMMARIZE EXAM_MARK PER ( EXAM_MARK { CourseId } ) :
                          { AvgMark := AVG ( mark ) }
```

Here the relation being summarized is the same as the relation providing the `PER` values. That is very commonly the case in practice, sufficiently so to perhaps warrant a further shorthand, and **Tutorial D** does in fact provide one in the form of `SUMMARIZE ... BY ...`, as illustrated in Example 5.11.

Example 5.11: Average mark for each exam, using SUMMARIZE … BY …

```
SUMMARIZE EXAM_MARK BY { CourseId } :
               { AvgMark := AVG ( mark ) }
```

In case the shorthands offered by `SUMMARIZE` … `BY` … hardly seem worth the addition to the language, consider that the `SUMMARIZE` operand can be a relation expression of any complexity. To be sure, writing it out twice is not much of a problem these days, thanks to copy-and-paste, but reading it twice and noticing the special case might be rather burdensome, both for human and computer (for Example 5.10 is likely to take significantly longer than Example 5.11 to compute unless the DBMS can notice that the `PER` operand in 5.10 is a projection of the `SUMMARIZE` operand).

Definition of `SUMMARIZE` … `BY` …

`SUMMARIZE` *r1* **`BY {`** *b1, …, bm* **`} : {`** *a1* **:=** *sum1, …, an* **:=** *sumn* **}**
where *r1* is a relation and *b1, …, bm* are names of attributes of *r1*, is equivalent to
`SUMMARIZE` *r1* `PER (` *r1* `{` *b1, …, bm* `} ) :`
`{` *a1* **:=** *sum1, …, an* := *sumn* `}`

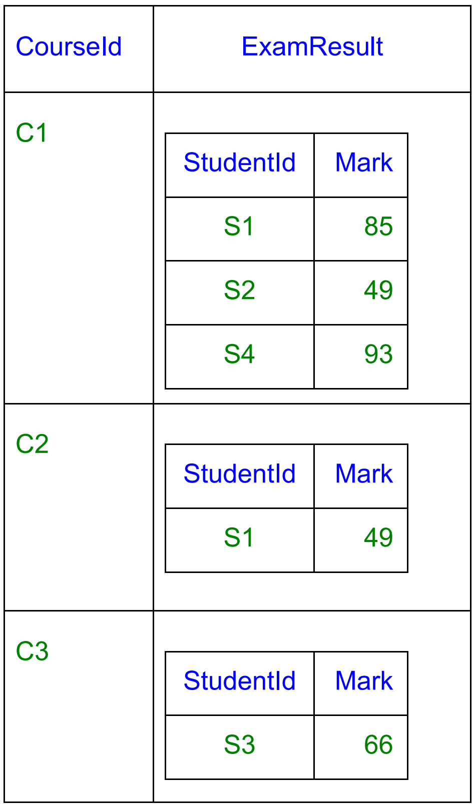
There remain just two more relational operators defined in **Tutorial D**. They both deal with nested relations and they come as a pair, `GROUP` and `UNGROUP`.

5.7 GROUP and UNGROUP

Example 5.8 includes the following expression, yielding an intermediate result named `C_ER2`:

```
EXTEND EXAM_MARK{CourseId}:
{ ExamResult := RELATION { TUPLE { CourseId CourseId } }
                COMPOSE EXAM_MARK }
```

Figure 5.6 shows the result. It differs from the `C_ER` of Figure 5.5 only in the absence of a tuple for course C4, whose exam nobody sat.



CourseId	ExamResult	
	StudentId	Mark
C1	S1	85
	S2	49
	S4	93
	StudentId	Mark
C2	S1	49
	StudentId	Mark
C3	S3	66

Figure 5.6: Intermediate result C_ER2 from Example 5.8

Tutorial D has a shorthand for the expression producing the relation shown in Figure 5.6. This shorthand uses a relational operator named `GROUP` and is illustrated in Example 5.12.

Example 5.12: Use of GROUP

```
EXAM_MARK GROUP { StudentId, Mark } AS ExamResult
```

Loosely speaking, Example 5.12 "groups" the `StudentId/Mark` combinations for each `CourseId` value, the result being a relation that becomes the value of the attribute `ExamResult` in the tuple for that `CourseId` value replacing the now redundant attributes `StudentId` and `Mark`. Somewhat less loosely, Example 5.12 pairs each `CourseId` value appearing in `EXAM_MARK` with its image relation in `EXAM_MARK`. (That is still a little loose, however. Recall that in general we speak of the image relation of a given *tuple*, not a given attribute value, in a given relation. Strictly speaking, then, for each `CourseId` value appearing in `EXAM_MARK` we take the tuple *t* consisting of just that attribute value and extend *t* with its image relation in `EXAM_MARK`.)

The specification `{ StudentId, Mark } AS ExamResult` is called a *grouping*.

Notice that Example 5.12, using `GROUP`, mentions `EXAM_MARK` only once, whereas Example 5.8 mentions it twice. Notice also that the first step in Example 5.8 is a projection of `EXAM_MARK` over `{CourseId}`. That projection is implied in Example 5.12, `CourseId` being the only attribute of `EXAM_MARK` that is not mentioned in the grouping for `ExamResult`.

Now, recall that in **Tutorial D**, wherever a commalist of attribute names is required, as in projection for example, that commalist can be preceded by the key words `ALL BUT` to indicate the attributes of the operand relation that are *not* to appear in the result. The following expression is therefore equivalent to Example 5.12:

```
        EXAM_MARK GROUP { ALL BUT CourseId } AS ExamResult
```

In case you are at all familiar with SQL I should now draw your attention to the fact that SQL has a counterpart of the `ALL BUT` method of specifying a grouping. This is its `GROUP BY` clause. In SQL, Example 5.12 would be expressed as

```
        FROM EXAM_MARK GROUP BY CourseId
```

but this is only a fragment and cannot appear as a complete expression. The complete expression in which the fragment appears cannot refer to the presumed column containing the nested tables because, as you can see, that column has no name. However, SQL does allow you to operate on those nested tables by aggregation, using constructs similar to those used for that purpose in invocations of **Tutorial D**'s `SUMMARIZE` operator.

Definition of GROUP

r **GROUP** { *al* } **AS** *g*, where:

- *r* is a relation such that the projection *r* **{** *al* **}** is defined, and
- *g* is an attribute name not appearing in the heading of *r* { ALL BUT *al* }

is equivalent to

```
( EXTEND r : { g := RELATION { t } COMPOSE r } )
{ ALL BUT al }
```

where *t* = TUPLE { *a1 a1*, …, *an an* }, where *a1*, …, *an* are the attributes of *r* { *al* }.

The inverse operator of `GROUP` is `UNGROUP`. Example 5.13 shows how to get back (so to speak) to `EXAM_MARK`, given `C_ER2` as the relation depicted in Figure 5.6.

Example 5.13: Use of UNGROUP

```
C_ER2 UNGROUP ExamResult
```

As you can see from Figures 5.4 and 5.1, ungrouping `C_ER` on `ExamResult` effectively replaces that attribute by its own attributes, `StudentId` and `Mark`, the tuples of the result being obtained from each tuple of `C_ER2` by joining its `CourseId` value with each tuple in turn of the `ExamResult` value.

In Example 5.13, the cardinality of the result is equal to the sum of the cardinalities of the `ExamResult` relations. A similar observation applies if we replace the operand `C_ER2` by the relation `C_ER` shown in Figure 5.5, because the extra tuple for course C4 contributes no tuples to the ungrouping result. *Exercise for the reader:* Is it *always* the case that the cardinality of an ungrouping is equal to the sum of the cardinalities of the relations the operand relation is being ungrouped on?

Definition of UNGROUP

The definition of UNGROUP uses an aggregate operator, UNION, that is not mentioned in Section 5.3 because its second operand is required to be a relation typed attribute. Recall from that chapter that UNION is commutative and associative, allowing us to defined the *n*-adic version, UNION { *r1, …, rn* }. The existence of that *n*-adic version allows us to define a corresponding aggregate operator. In UNION (*r, x*) the second operand, whose declared type must be a relation type, provides the relations *r1, … rn* for an equivalent invocation of the *n*-adic operator. Here, then, is the definition of UNGROUP:

r **UNGROUP** *a*, where:

- *r* is a relation, *a* is an attribute of *r*, and the declared type *ta* of *a* is a relation type
- no attribute of *ta* has an attribute name used in the heading of *r*, except perhaps *a* itself.

is equivalent to

```
          UNION ( EXTEND r : { x := RELATION { t } TIMES a },
             x )
```

where *x* is an attribute name arbitrarily chosen and *t* = TUPLE { *b1 b1, …, bn bn* }, where *b1, …, bn* are the attributes of *r*{ALL BUT *a*}.

As with GROUP, Version 1 of **Tutorial D** allows the second operand to be a commalist, thus allowing more than one relation-valued attribute to be specified in a single invocation. Version 2 allows just a single attribute and you are advised against using more than one attribute with Version 1.

Points to note:

- The invocation of TIMES used in the extend addition effectively extends the (relation) value of *a* in each tuple of *r* with the other attribute values of the same tuple. The resulting relation is the value of the attribute *x* by which that tuple of *r* is extended.
- The aggregate union of all the *x* values is the result of the invocation of UNGROUP.
- Although UNGROUP is the inverse operator for GROUP, GROUP is not the inverse of UNGROUP. In other words, although *r* = (*r* GROUP { *al* } AS *g*) UNGROUP *g* in general, it is not necessarily the case that *s* = (*s* UNGROUP *g*) GROUP { *al* } AS g, where *al* specifies the attributes of the relation-valued attribute *g*. Some tuples in *s* might have an empty relation as the value for *g*, whereas an invocation of GROUP can never give rise to such empty relations. Also, several distinct tuples might appear in *s* that differ only in their *g* values, in which case those tuples will be "condensed" into a single tuple in the grouping.

5.8 WRAP and UNWRAP

It is occasionally convenient to be able to collect together certain attribute values in each tuple of a relation to form a single attribute value that is itself a tuple, replacing the collected values. Example 5.14 shows how this effect can be achieved, laboriously, using `EXTEND` and projection on a relvar named `CONTACT_INFO`.

Example 5.14: Collecting attribute values together

```
CONTACT_INFO WRAP { House, Street, City, Zip } AS Address
```
Assuming that is assigned to a relvar `CONTACT_INFO_WRAPPED`:
```
CONTACT_INFO_WRAPPED UNWRAP Address
```

The repetitions of the attribute names involved suggest a shorthand for this purpose and in **Tutorial D** it appears as the relational operator `WRAP`, which is accompanied in the language by its inverse, `UNWRAP`. In Example 5.15 `WRAP` is used to collect together the components of people's postal addresses to form the tuple-valued attribute `Address`, and `UNWRAP` is used to reverse the process.

Example 5.15: Use of WRAP and UNWRAP

```
CONTACT_INFO WRAP { House, Street, City, Zip } AS Address
```
Assuming that is assigned to a relvar `CONTACT_INFO_WRAPPED`:
```
CONTACT_INFO_WRAPPED UNWRAP ( Address )
```

Definitions of WRAP and UNWRAP

r **`WRAP`** **{** *al* **}** **`AS`** *w***,** where:

- *r* is a relation such that the projection *r* **{** *al* **}** is defined, and
- *w* is an attribute name not used in the heading of *r*

is equivalent to

```
( EXTEND r : { w := t } ) { ALL BUT al }
```

where *t* = `TUPLE { b1 b1, …, bn bn }`, where *b1, …, bn* are the attributes of *r* `{ al }`.

r **`UNWRAP`** *a***,** where:

- *r* is a relation, *a* is an attribute of *r*, and the declared type *ta* of *a* is a tuple type
- no attribute of *ta* has an attribute name used in the heading of *r*, except perhaps *a* itself.

is equivalent to

```
( EXTEND r : { b1 := b1 FROM a, …, b1 := bn FROM a } )
{ ALL BUT a }
```

where *b1, …, bn* are the attributes of *ta*.

An expression of the form *a* `FROM` *t1* is called *attribute extraction*, defined in Section 5.10. It denotes the value of attribute *a* of tuple *t1*.

You have now met nearly all of the relational operators of **Tutorial D** Version 1 (there are a couple more that have been deliberately omitted as beyond the scope of an introduction to the subject). Here they are again, in categories monadic, dyadic, and *n*-adic according to their number of relation operands.

Monadic:	RENAME, projection, WHERE (restriction), EXTEND, SUMMARIZE … BY, GROUP, UNGROUP, WRAP, UNWRAP
Dyadic:	JOIN, UNION, INTERSECT, NOT MATCHING (semidifference), MINUS (difference), MATCHING (semijoin), COMPOSE, SUMMARIZE … PER
***n*-adic:**	JOIN { … }, UNION { … }, INTERSECT { … }

There remain to be described various non-relational operators that involve tuples or relations and are defined in **Tutorial D**, being deemed useful additional ingredients of a relational database language.

5.9 Relation Comparison

The operators described in this section are especially useful for defining database constraints, as described in Chapter 6, but they can be useful in queries too.

You are familiar with comparisons: dyadic, truth-valued or Boolean operators whose operands are of the same type. For example, comparisons of the form *x* = *y*, where *x* and *y* are expressions of the same type, are available for all types in **Tutorial D**, as you would surely expect. However, some computer languages do not support "=" for all the types they recognize, and some do not support it correctly!—i.e., in the strict sense that is needed for relational databases.

A Note on Equality

In **Tutorial D**, the literals `TRUE` and `FALSE` denote the only two values of the type named `BOOLEAN`, commonly called truth values. The comparison *x* = *y* yields `TRUE` if the expressions *x* and *y* denote the same value; otherwise (they denote different values) it yields `FALSE`. That is the strict sense I just mentioned. As a consequence, if an expression *w* contains one or more appearances of *x* and we obtain expression *w'* from *w* by replacing every appearance of *x* by *y*, then *w* = *w'* has the same truth value as *x* = *y*. Some languages, such as COBOL and SQL, deviate somewhat from this strict definition of equality. In particular, those two languages both allow two character strings to "compare equal" if they differ only in their numbers of trailing blanks—for example, the strings `'this'` and `'this  '`. Such treatment is disastrous in a relational database language because the DBMS relies on the strict sense of "=" for the definition and implementation of so many of the operators described in this book.

Suppose, for example, that in the current value of `IS_CALLED` one of the two Borises had his name recorded with a trailing blank, and **Tutorial D**'s definition of "=" were the same as SQL's and COBOL's. What then would be the result of the projection `IS_CALLED{Name}`? It can't include both of `TUPLE{Name NAME('Boris')}` and `TUPLE{Name NAME('Boris ')}`, for those two tuples would be deemed equal and thus cannot both appear in the same relation. And if only one of them can appear, which one? In fact, does it have to be either of them? Couldn't `TUPLE{Name NAME('Boris '})`, with two trailing blanks, appear instead? Similar questions arise in connection with Example 4.3 in Chapter 4, where `Name` is the common attribute for an invocation of `JOIN`.

A language that allows the declared type of an attribute to be one for which "=" is not supported (for example, SQL) is relationally incomplete. If, for example, relation *r* has such an attribute, *a,* then no projection of *r* that includes *a* can be defined. One such projection is the identity projection, *r*`{ALL BUT}`, and if that is undefined it is difficult to see how even the expression *r* can be defined!

It follows in particular from the foregoing discussion that **Tutorial D**'s support of $x = y$ allows x and y to denote relations (of the same type). **Tutorial D** also supports relation comparisons of the form $r1 \subseteq r2$ ("*r1 is a subset of r2*") and its inverse, $r1 \supseteq r2$ ("*r1 is a superset of r2*").

Definitions of relation comparison

Let *r1* and *r2* be relations having the same heading. Then:

$r1 \subseteq r2$ is *true* if every tuple of *r1* is also a tuple of *r2*, otherwise *false*.
$r1 \supseteq r2$ is equivalent to $r2 \subseteq r1$
$r1 = r2$ is equivalent to $r1 \subseteq r2$ AND $r2 \subseteq r1$

Note carefully that the symbols ⊆ and ⊇ are often referred to "subset of or equal to" and "superset of or equal to", respectively. The words "or equal to" are added for clarity only—they are redundant because by definition every set is a subset of itself. Note that only one relation comparison operator needs to be taken as primitive, either ⊆ or ⊇, for the other two can than be defined in terms of it.

***REL* Alert**

Because the mathematical symbols ⊆ and ⊇ are unlikely to be easily available on your keyboard, *Rel* allows you to use the combinations <= and >=, respectively, in their places. These combinations are also used for "less than or equal to" and "greater than or equal to", so you have to read *Rel* expressions carefully to avoid confusion. There is no ambiguity, because there are no types in **Tutorial D** for which both "less than" and "subset of" are defined.

The alert reader will have noticed that the definitions of relation comparisons tacitly depend on a definition of *tuple* equality. To determine whether tuple *t* appears in the bodies of both *r1* and *r2* the system must know how to evaluate *t1* = *t2* where *t1* and *t2* are tuples.

Definition of tuple equality

Let *t1* and *t2* be tuples having the same heading. Then:
t1 = *t2* is *true* if for every attribute *a* of *t1*, *a* `FROM` *t1* = *a* `FROM` *t2*; otherwise it is *false*.

The expression *a* `FROM` *t1* is an example of "attribute extraction", as already mentioned in connection with the relational operator `UNWRAP` in Section 5.8. Just as relation equality depends on tuple equality, tuple equality in turn depends on equality being defined for all types. In fact the definition is recursive, because the declared type of an attribute can be a tuple type or a relation type.

Now, consider the relation comparison

```
r { } = TABLE_DUM
```

which is clearly defined for all relations *r*. Did you see immediately that it evaluates to `TRUE` if and only if *r* is empty? For if *r* is empty, then so is every projection of *r*, and if *r* is not empty, then nor is any projection of *r*. `TABLE_DUM`, recall, is **Tutorial D**'s pet name for the empty relation of degree zero. Well, recognizing that taking a projection and comparing the result with an empty relation might strike some people as a long-winded and not very obvious way of testing a relation for being empty, **Tutorial D** provides the shorthand

```
IS_EMPTY(r)
```

as being equivalent to that comparison (and also to `COUNT(`*r*`)=0`, of course).

Uses for Relation Comparisons

As I have already suggested, relation comparisons are mostly used in the definition of database constraints. Their use for that purpose is described in the next chapter. Here I give just one example of the use of relational comparison in a query.

Suppose we wish to discover which students have taken the exam for every course on which they are enrolled. In that case we need the relation representing the predicate

> For every course *CourseId* on which student *StudentId*, who is called *Name*, is enrolled, there exists a mark *Mark* such that *StudentId* scored *Mark* on the exam for *CourseId*.

That predicate has just two parameters, *StudentId* and *Name.* The other variables, *CourseId* and *Mark* are both quantified and therefore bound. *Mark* is existentially quantified, suggesting the use of projection on EXAM_MARK, but *CourseId* is universally quantified. I haven't given you a relational operator corresponding to universal quantification and in fact **Tutorial D** doesn't have one. (It did, once, but the operator in question, named DIVIDEBY, turned out to be somewhat troublesome and difficult to use and is now deprecated.) However, universal quantification can be expressed, albeit in an unpleasantly roundabout way, using existential quantification and negation. The students who have sat the exam for every course they are enrolled on are exactly those students for whom there does not exist a course, on which they are enrolled, whose exam they have not sat. The double negation used in that sentence shows up in Example 5.16 as two invocations of NOT MATCHING.

Example 5.16: Students who have taken the exam for every course they are enrolled on

```
IS_CALLED NOT MATCHING (IS_ENROLLED_ON NOT MATCHING EXAM_MARK)
```

Explanation 5.16:

- **IS_ENROLLED_ON NOT MATCHING EXAM_MARK** gives the relation consisting of those tuples of IS_ENROLLED_ON that have no matching tuple in EXAM_MARK. In other words, those tuples that satisfy the predicate "Student *StudentId* is enrolled on course *CourseId* and there does not exist a mark *Mark* such that *StudentId* scored *Mark* in the exam for *CourseId*." The projection representing the existential quantification of *Mark* here is not explicitly given in Example 5.16 but is implicit in the use of NOT MATCHING: IS_ENROLLED_ON NOT MATCHING EXAM_MARK is equivalent to IS_ENROLLED_ON MINUS (EXAM_MARK{ALL BUT Mark}), where the projection does appear explicitly.
- **IS_CALLED NOT MATCHING (** IS_ENROLLED_ON NOT MATCHING EXAM_MARK **)** gives the relation consisting of those tuples of IS_CALLED that have no matching tuple in the relation representing enrolments for which there is no matching exam result. Those tuples are precisely the ones that satisfy our predicate. Note that a tuple for student S5, who is enrolled on no courses at all, correctly appears in the result. That's because there does not exist a course for which S5 is enrolled but has not taken the exam.

But if you find double negation a bit much to get your head around, you might prefer the alternative given in Example 5.17.

Example 5.17: Alternative solution to Example 5.16 using ⊆

```
IS_CALLED WHERE
IS_ENROLLED_ON COMPOSE RELATION {TUPLE {StudentId StudentId}}
⊆
( EXAM_MARK COMPOSE RELATION {TUPLE {StudentId StudentId}} )
 {ALL BUT Mark}
```

Explanation 5.17:

- **`IS_CALLED WHERE`** announces clearly that the result of our query is a relation whose body is a subset of that of `IS_CALLED`; in other words, we are looking for just those students that have the particular property defined in the `WHERE` condition.
- The particular property defined in the `WHERE` condition is such that the entire query translates roughly to students whose every enrolment is on a course for which they took the exam.
- The relations being compared in the `WHERE` condition are the image relations of `IS_CALLED` tuples in `IS_ENROLLED_ON` and `EXAM_MARK` minus the `Mark` attribute. The commonality between the somewhat cumbersome expressions denoting those image relations suggests that some shorthand embracing that commonality would be both feasible and useful. The commonality is the invocation of `COMPOSE` with a singleton relation consisting of a tuple derived from the relation operand of `WHERE`. Under a suggestion from Chris Date in references [10] and [11] the fragment *r* `COMPOSE RELATION {TUPLE {StudentId StudentId}}` is reduced to just ‼*r* (where ‼ is the double exclamation mark, sometimes pronounced "bang bang"), like this:
  `IS_CALLED WHERE ‼IS_ENROLLED_ON ⊆ ‼(EXAM_MARK{ALL BUT Mark})`

5.10 Other Operators on Relations and Tuples

We close this chapter with brief descriptions of other operators defined in **Tutorial D** that operate on relations or tuples.

Tuple Membership Test

Let *r* be a relation and let *t* be a tuple of the same heading as *r*. Then

```
t ∈ r
```

is defined to yield `TRUE` if the body of *r* contains tuple *t*, otherwise `FALSE`.

***REL* Alert**

Because the mathematical symbol ∊ is unlikely to be easily available on your keyboard, *Rel* allows you to use the key word IN in its place.

Tuple Extraction

Let *r* be a relation of cardinality one (a "singleton relation"). Then

```
TUPLE FROM r
```

is defined to yield the single tuple contained in the body of *r*. For example,

```
TUPLE FROM COURSE WHERE CourseId = CID('C1')
```

yielding `TUPLE { CourseId CID('C1'), Title 'Database' }`.

Attribute Value Extraction (previously mentioned in Section 5.8)

Let *t* be a tuple with an attribute named *a*. Then

```
a FROM t
```

is defined to yield the value of the attribute *a* in tuple *t*. For example,

```
Title FROM TUPLE FROM COURSE WHERE CourseId = CID('C1')
```

yielding the `CHAR` value `'Database'`.

Tuple Counterparts of Relational Operators

Let *t1* and *t2* be tuples. Then the following are defined, with obvious semantics in each case and in each case yielding a tuple:

- tuple rename:

  t1 `RENAME {` *a1* `AS` *b1*, …, *an* `AS` *bn* `}`

- tuple projection:

  t1 `{ [ALL BUT]` *a1*, … *an* `}`

- tuple extension:
  `EXTEND` *t1* `: {` *a1* `:=` *exp1*, …, *a1* `:=` *expn* `}`

- tuple compose:
  t1 `COMPOSE` *t2*

- tuple union:
  t1 `UNION` *t2*

- tuple wrap:
  t1 `WRAP { `*a1*, …, *an*` } AS` *a*

- tuple unwrap:
  t1 `UNWRAP` *a*

EXERCISES

1. (Repeated from the body of the chapter) What can you say about the result of *r1* `COMPOSE` *r2* when *r1* and *r2* have identical headings? For example, what is the result of `IS_CALLED COMPOSE IS_CALLED`?
2. (Repeated from the body of the chapter) Is `COMPOSE` associative? In other words, is (*r1* `COMPOSE` *r2*) `COMPOSE` *r3* equivalent to *r1* `COMPOSE` (*r2* `COMPOSE` *r3*)? If so, prove it; if not, show why.
3. What can you say about the result of *r1* `MATCHING` (*r2* `MATCHING` *r1*)?
4. (Repeated from the body of the chapter) Does the aggregate operator `AVG` have a basis operator? If so, define it.
5. Suppose an aggregate operator `PRODUCT` is defined, with arithmetic multiplication as its basis operator. What is the result of `PRODUCT(`*r,x*`)` if *r* is empty?
6. (Repeated from the body of the chapter) Is it *always* the case that the cardinality of an ungrouping is equal to the sum of the cardinalities of the relations being ungrouped on?
7. Write **Tutorial D** expressions for the following queries and get *Rel* to evaluate them:
   a) Get the total number of parts supplied by supplier S1.
   b) Get supplier numbers for suppliers whose city is first in the alphabetic list of such cities.
   c) Get part numbers for parts supplied by all suppliers in London.
   d) Get supplier numbers and names for suppliers who supply all the purple parts.
   e) Get all pairs of supplier numbers, S*x* and S*y* say, such that S*x* and S*y* supply exactly the same set of parts each.
   f) Write a truth-valued expression to determine whether all supplier names are unique in `S`.
   g) Write a truth-valued expression to determine whether all part numbers appearing in `SP` also appear in `P`.

6 Constraints and Updating

6.1 Introduction

You have already met constraints, in type definitions (Chapter 2), where they are used to define the set of values constituting a type. The major part of this chapter is about *database* constraints. Database constraints express the integrity rules that apply to the database. They express these rules to the DBMS. By enforcing them, the DBMS ensures that the database is at all times *consistent* with respect to those rules.

In Chapter 1, Example 1.3, you saw a simple example of a database constraint declaration expressed in **Tutorial D**, repeated here as Example 6.1 (though now referencing IS_ENROLLED_ON rather than ENROLMENT).

Example 6.1: Declaring an integrity constraint

```
CONSTRAINT MAX_ENROLMENTS
     COUNT ( IS_ENROLLED_ON ) ⩾ 20000 ;
```

The first line tells the DBMS that a constraint named `MAX_ENROLMENTS` is being declared. The second line gives the expression to be evaluated whenever the DBMS decides to check that constraint. This particular constraint expresses a rule to the effect that there can never be more than 20000 enrolments altogether. It is perhaps an unrealistic rule and it was chosen in Chapter 1 for its simplicity. Now that you have learned the operators described in Chapters 4 and 5 you have all the equipment you need to express more complicated constraints and more typical ones. This chapter explains how to use those operators for that purpose.

Now, if a database is currently consistent with its declared constraints, then there is clearly no need for the DBMS to test its consistency again until either some new constraint is declared to the DBMS, or, more likely, the database is *updated*. For that reason, it is also appropriate in this chapter to deal with methods of updating the database, for it is not a bad idea to think about which kinds of constraints might be violated by which kinds of updating operations, as we shall see.

6.2 A Closer Look at Constraints and Consistency

A constraint is defined by a truth-valued expression, such as a comparison. A database constraint is defined by a truth-valued expression that references the database. To be precise, the expression defines a *condition* that must be *satisfied* by the database at all times. We have previously used such terminology in connection with tuples—in relational restriction for example, which yields a relation containing just those tuples of a given relation that satisfy the given condition. We can justify the use of the terminology in connection with database constraints by considering the database value (or "state", as it is sometimes called) at any particular point in time to be a tuple. The attributes of this tuple take their names and declared types from the variables constituting the database and their values are the values of those variables. Taking this view, the database itself is a tuple variable and every successful update operation conceptually assigns a tuple value to that variable, even if it actually assigns just one relation value to one relation variable, leaving the other relvars unchanged.

When Are Constraints Checked?

What do we really mean when we say that the DBMS must ensure that the database is consistent *at all times*? Internally, the DBMS might have to perform several disk writes to complete what is perceived by the user as a single update operation, but intermediate states arising during this process are visible to nobody (or should be so—certain commercially available systems that call themselves DBMSs nevertheless do allow such intermediate states to be visible). Because those intermediate states are invisible, we can state that if the database is guaranteed to be consistent immediately following completion of each single statement that updates it, then it will be consistent whenever it is visible. We say therefore that, conceptually at least, constraints are checked at all *statement boundaries,* and only at statement boundaries—we don't care about the consistency of intermediate states arising during the DBMS's processing of a statement because those states aren't visible to us in any case.

To clarify "all statement boundaries", first, note that this includes statements that are contained inside other statements, such as `IF` … `THEN` … `ELSE` … constructs for example. Secondly, the conceptual checking need not take place at all for a statement that does no updating, but no harm is done to our model if we *think* of constraints as being checked at every statement boundary.

In **Tutorial D**, as in many computer languages, a statement boundary is denoted by a semicolon, so we can usefully think of constraints as being effectively checked at every semicolon. If all the constraints are satisfied, then the updates brought about by the statement just completed are accepted and made visible; on the other hand, if some constraint is not satisfied, then the updates are rejected and the database reverts to the value it had immediately after the most recent successful statement execution.

Declared Constraints and *The* Database Constraint

We can usually expect a database to be subject to quite a few separately declared constraints. To say that the database must satisfy all of the conditions specified by these constraints is equivalent to saying that it must satisfy the single condition that is the *conjunction* of those individually specified conditions—the condition formed by connecting them all together using logical `AND`. We can conveniently refer to the resulting condition as *the* database constraint. Now we can state the principle governing correct maintenance of database integrity by the DBMS quite succinctly: the database constraint is guaranteed to be satisfied at every statement boundary.

6.3 Expressing Constraint Conditions

Use of Relational Operators

The condition for a database constraint must reference the database and therefore must mention at least one variable in that database. In the case of relational databases, that means that at least one relvar must be mentioned. Moreover, as the condition is specified by a single expression (a truth-valued expression), it must use relational operators if it involves more than one relvar and, as we shall soon see, is likely to use them even when it involves just one relvar.

However, a relation isn't a truth value, so we need some of the non-relational operators described in Chapter 5, in addition to the relational operators, to express conditions for declared constraints. In particular, the expression itself must denote an invocation of some truth-valued operator. In Example 6.1 that operator is "=". No relational operators are used in that example, because the only relation we need to operate on is the one that is the value of the relvar `IS_ENROLLED_ON` when the constraint is checked. The aggregate operator `COUNT` operates on that relation to give its cardinality, an integer.

Use of `COUNT` and `IS_EMPTY`

It turns out that if the database language is relationally complete and also supports `COUNT`, along with the usual numerical comparison operators, then it can also be regarded as complete for the purpose of expressing constraints (so long as the support for `COUNT` is orthogonal, such that an invocation of it can appear wherever an integer literal would be permitted). Furthermore, every constraint can be expressed as a single comparison, one of whose operands is an invocation of `COUNT`. It seems, then, that the only operators we need in addition to those required for relational completeness are ones that we would surely have anyway for use in queries. However, requiring every constraint to be expressed as a comparison involving `COUNT` would not be very kind to users of our language. We need to explore the possibilities for more convenient ways of expressing common kinds of constraint.

One particular kind of comparison involving `COUNT` is an expression of the form `COUNT(`*r*`)=0`, where *r* denotes a relation. This is effectively a test for emptiness on *r*. If *r* is empty, then there does not exist a tuple that satisfies the predicate for *r*. If, on the other hand, there does exist at least one such tuple, then it is a tuple that in a manner of speaking breaks the rule expressed by the constraint. So, if we can write a relational expression denoting the relation whose body consists of all the tuples representing counterexamples to the rule in question, then we can enforce that rule by requiring the cardinality of that relation to be zero. And that method of expressing a constraint turns out to be sufficient for *any* constraint that might be required, including even Example 6.1. But first look at Example 6.2 for an illustration of the idea. It enforces a rule to the effect that every student sitting an exam must be enrolled on the relevant course.

Example 6.2: Testing for absence of counterexamples

```
CONSTRAINT Must_be_enrolled_to_take_exam
     COUNT ( EXAM_MARK NOT MATCHING IS_ENROLLED_ON ) = 0 ;
```

The expression `EXAM_MARK NOT MATCHING IS_ENROLLED_ON` denotes the relation whose body consists of those tuples of `EXAM_MARK` that have no matching tuple (on the common attributes `StudentId` and `CourseId`) in `IS_ENROLLED_ON`, and we don't want there ever to be any such tuples. But counting all the tuples in a relation and then seeing if the result is zero is a rather heavy-handed way of expressing what to a logician is nothing more than an existence test, negated. That is why **Tutorial D** provides the shorthand `IS_EMPTY(`*r*`)` for `COUNT(`*r*`)=0`, as shown in Example 6.3.

Example 6.3: Use of IS_EMPTY

```
CONSTRAINT Must_be_enrolled_to_take_exam_alternative1
     IS_EMPTY ( EXAM_MARK NOT MATCHING IS_ENROLLED_ON ) ;
```

In case you are now wondering how the constraint in Example 6.1 can be expressed as a single invocation of `IS_EMPTY`, and thus questioning my claim that every constraint that can be expressed according to the theory can be expressed as a test for zero cardinality, Example 6.4 shows you one way of doing it, but note carefully that the expression still involves `COUNT`.

Example 6.4: MAX_ENROLMENTS expressed as an invocation of IS_EMPTY

```
CONSTRAINT MAX_ENROLMENTS_alternative1
  IS_EMPTY ( RELATION { TUPLE { N COUNT(IS_ENROLLED_ON) } }
             WHERE N > 20000 ) ;
```

And here, of course, it is the invocation of `IS_EMPTY` that is significantly more "heavy-handed" than the simple comparison used in Example 6.1—all I have done, in fact, is to bury that comparison as a restriction condition in a somewhat contrived relational expression.

Explanation 6.4

- **RELATION { TUPLE { N COUNT(IS_ENROLLED_ON) } }** denotes the relation of heading `{ N INTEGER }` in whose single tuple the value of the attribute `N` is the number of tuples in the current value of `IS_ENROLLED_ON`.
- **WHERE N > 20000** operates on that singleton relation to yield the empty relation of heading `{ N INTEGER }` if and only if `TUPLE { N COUNT(IS_ENROLLED_ON) }` fails to satisfy the condition `N > 20000`. Thus, the result is empty only when the number of enrolments is in fact no greater than the maximum allowed.

As this example shows, any value of any type can be "converted" to a tuple of degree one by invocation of the tuple selector, and the resulting tuple can be "converted" to a relation of cardinality one by invocation of the relation selector. The technique quite often turns out to be useful.

Now, knowing that a language is relationally complete gives us a clear understanding of one very important aspect of its expressive power, but for the full picture we need to know what built-in types it supports in addition to relation types and `BOOLEAN`, and what operators are available for operating on values of those types. If we can assume the availability of `IS_EMPTY`, which operates on a relation to yield a truth value, then we have a theoretically satisfying notion of completeness for expressing constraints. If the language is relationally complete, then every constraint can be expressed in the form `IS_EMPTY(`*r*`)` and the expressive power of the language for defining constraints depends on what additional types are available to be declared types of relation attributes. However, it turns out that in **Tutorial D** anything that can be expressed in the form `IS_EMPTY(`*r*`)` can be expressed in several other ways too, as I am about to describe, and it is a moot point which of these several methods the theoretician might consider to be the most satisfying.

Use of Relation Comparisons

Relational comparisons are described in Chapter 5, Section 5.9. It turns out that every constraint that can be expressed using `IS_EMPTY` can be expressed as a single comparison of the form *r1* ⊆ *r2*, where *r1* and *r2* are relations, which is true if and only if the body of *r1* is a subset of that of *r2*. Example 6.5 shows how "⊆" provides an alternative way of expressing the constraint declared in Example 6.2, requiring every student who taking an exam to be enrolled on the relevant course.

Example 6.5: Use of ⊆

```
CONSTRAINT Must_be_enrolled_to_take_exam_alternative1
     EXAM_MARK { StudentId, CourseId } ⊆
     IS_ENROLLED_ON { StudentId, CourseId } ;
```

This might be considered to be clearer than Example 6.3 but it needs to name the common attributes in the projections needed to obtain relations of the same type for the comparison (though in this particular example the projection of `IS_ENROLLED_ON` is the identity projection, over the entire heading, and so can be omitted).

In case you are wondering how Example 6.1 might be expressed using ⊆, Example 6.6 shows one rather straightforward way of doing it, as well as perhaps giving a compelling reason why we might prefer not to be compelled to express every constraint in the form *r1* ⊆ *r2*.

Example 6.6: MAX_ENROLMENTS expressed as a relation comparison

```
CONSTRAINT MAX_ENROLMENTS_REV1
 RELATION { TUPLE { N COUNT(IS_ENROLLED_ON) } } WHERE N > 20000
 ⊆ RELATION { N INTEGER } { } ;
```

Explanation 6.6

- The explanation of the first line is as in Example 6.4. The resulting relation is the first operand of an invocation of "⊆".
- **`RELATION { N INTEGER } { }`** is the second operand, denoting the empty relation of the same type as as the first operand. Of course, to be a subset of an empty relation is the same as to *be* (i.e., be equal to) that empty relation, so the invocation of "⊆" yields `TRUE` if and only if the first operand is in fact that empty relation.

Obviously, wherever we can use "⊆" we could instead use "⊇", from the equivalence of *r1* ⊆ *r2* and *r2* ⊇ *r2*. Moreover, the given explanation of Example 6.6 clearly shows that "=" comparison can be used to express every constraint that can be expressed in the form `IS_EMPTY(`*r*`)` (which could also be written as *r* = *r* `WHERE FALSE`). But the availability of "=" comparisons on values of all types, including relations in particular, is surely required for relational completeness. Under that assumption we could argue that relational completeness is *all* that is theoretically needed for complete support for constraints.

Now, I claimed that every constraint can be expressed as a *single* comparison of the form *r1* ⊆ *r2*. You might be wondering how equality of relations can be expressed using a single invocation of "⊆". Clearly, *r1* = *r2* is equivalent to *r1* ⊆ *r2* `AND` *r2* ⊆ *r1*, but that expression is an invocation of `AND`, not "⊆". Example 6.7 shows one way of doing it with a single invocation of "⊆".

Example 6.7: Relation equality using a single invocation of "⊆"

```
( ( r1 MINUS r2 ) UNION ( r2 MINUS r1 ) ) { } ⊆ RELATION { } { }
```

Explanation 6.7

- Recall: *r1* `MINUS` *r2* is equivalent to *r1* `NOT MATCHING` *r2* but requires *r1* and *r2* to be relations of the same type.
- **`( ( r1 MINUS r2 ) UNION ( r2 MINUS r1 ) )`** yields the relation whose body consists of every tuple of *r1* that is not also a tuple of *r2* and every tuple of *r2* that is not also a tuple of *r1*. This is sometimes called the *symmetric difference* of *r1* and *r2* (and a relational language might well provide a dyadic operator as a shorthand for expressing it). Note that the symmetric difference of sets *A* and *B* is the empty set if and only if *A*=*B* (i.e., they are one and the same set).
- Noting that a projection of relation *r* is empty if and only if *r* itself is empty, we can test the symmetric difference for being empty by taking its projection over no attributes and testing that projection for being a subset of the empty relation of degree zero (recall that in **Tutorial D** you can use the name TABLE_DUM for this relation if you prefer).

To prove that every expression of the form `IS_EMPTY(`*r*`)` is equivalent to some expression of the form *r1* ⊆ *r2* I merely note that `IS_EMPTY(`*r*`)` is equivalent to *r* ⊆ `(` *r* `WHERE FALSE )`. And as *r1* ⊆ *r2* is equivalent to `IS_EMPTY(`*r1* `MINUS` *r2*`)` it is clear that a language can support either `IS_EMPTY` or just one of our three relation comparison operators with equal expressive power. That gives four choices, so far, for the operator that allows us to express any theoretically expressible constraint as a single invocation of that operator on one or two relations. There are more!

Use of Truth-Valued Aggregate Operators

Our relvar `EXAM_MARK` really ought to be subject to a constraint requiring every value for the `Mark` attribute to lie in the range 0 to 100. That is easy enough to express using `IS_EMPTY`, as Example 6.8 shows, but many people would prefer to say that every mark shall lie within the required range instead of saying that no mark shall lie outside it.

Example 6.8: Restricting exam marks to between 0 and 100

```
CONSTRAINT Marks_between_0_and_100
     IS_EMPTY ( EXAM_MARK WHERE Mark ⩾ 0 OR Mark ⩽ 100 ) ;
```

In Chapter 5 you met the aggregate operator `AND`, named after its own basis operator. `AND(`*r,c*`)`, where *r* is a relation and *c* is a condition, is true if and only if every tuple of *r* satisfies *c*. (In *Rel* `ALL` is a synonym for aggregate `AND`. You might find `ALL(`*r,c*`)` more intuitive than `AND(`*r,c*`)`.) Use of aggregate `AND` allows many constraints to be expressed more succinctly and more clearly than use of any of the other methods we have met so far, as Example 6.9 shows in the case of our constraint on exam marks. Note, however, that this example depends on an enhancement in Version 2 of **Tutorial D**, as described in Chapter 5, Section 5.3, **Aggregate Operators**.

Example 6.9: Restricting exam marks to between 0 and 100 using aggregate AND as supported in Version 2 of **Tutorial D**

```
CONSTRAINT Marks_between_0_and_100_using_AND
     AND ( EXAM_MARK, Mark ⩾ 0 AND Mark ⩽ 100 ) ;
```

To show that aggregate `AND` is in fact yet another candidate for our single additional operator, and that in fact every constraint can be expressed as an invocation of that operator, I note the equivalence of `IS_EMPTY(`*r*`)` and `AND(`*r*`, FALSE)`. No tuple satisfies the condition `FALSE`, so `AND(`*r*`, FALSE)` is false whenever *r* contains at least one tuple and is true only when *r* is empty.

We have aggregate `OR` too, so, recalling from Chapter 2 that "for all *x*, *p*(*x*)" is equivalent to "there does not exist *x* such that `NOT(`*p*(*x*)`)`", we can note that `AND(`*r*, *c*`)` is equivalent to `NOT(OR(`*r*, `NOT(`*c*`)))`, from which it follows that `AND(`*r*, `FALSE)` is equivalent to `NOT(OR(`*r*, `TRUE))`. Finally, as `OR(`*r*, `TRUE)` is false only when *r* is empty, we can note that `OR(`*r*, `TRUE)` is equivalent to `NOT(IS_EMPTY(`*r*`))`.

Faced with such a plethora of choice for general methods of expressing constraints, **Tutorial D** does not arbitrate in favour of any of the noted candidates, allowing the user to choose freely from among them whichever is deemed most suitable for each particular purpose. The availability of logical connectives gives the user the further freedom to decide how best to arrange *the* database constraint into declared constraints, individually named and formulated. Sadly, we cannot say the same for the commercially available DBMSs at the time of writing (2014), for we are not aware of any widely available SQL implementation that supports *any* of the noted candidates for use in constraints: they don't support the international standard's `CREATE ASSERTION` statement, which would be SQL's counterpart of **Tutorial D**'s `CONSTRAINT` statement, and they don't permit table expressions to appear inside SQL's so-called "table constraints". Typically, the SQL user is restricted to certain special-purpose shorthands of the kinds described in the next section.

6.4 Useful Shorthands for Expressing Constraints

In Chapter 5 I showed how a relational database language can be extended by defining new relational operators—“shorthands”—in terms of the existing ones. If the existing language is relationally complete, then such extensions do not increase the language’s expressive power—there is no need for that—but, judiciously chosen, they do make some problems easier to solve by providing shorthands that are not only convenient but, by raising the level of abstraction, might also be easier to understand than the longhands on which they are defined. In Chapter 5 I illustrated this point by showing you the handful of such operators that have been “judiciously chosen” for **Tutorial D**, these having been proposed by various writers over the years. Unfortunately, very little in the way of useful shorthands has been proposed for use in constraints; and what little there is is subject to a certain amount of controversy. Yet the requirement for shorthands seems to be compelling, not just for the convenience of users but also for *performance,* as I will now explain.

Suppose that we require every constraint to be expressed using an expression of the form `IS_EMPTY(`*r*`)`. Then consider a simple constraint such as the one to make sure every exam mark is in the range of 0 to 100 and assume it is expressed as shown in either of Examples 6.8 and 6.9. Suppose that a certain update statement is used to add a single tuple to `EXAM_MARK`. Whether `IS_EMPTY` or aggregate `AND` is chosen for the constraint declaration, a naïve evaluation would involve the system in examining each existing `EXAM_MARK` tuple as well as the one being added. But the existing tuples are all known to satisfy the condition `Mark ≥ 0 AND Mark ≤ 100`, for if one of them didn’t the database would have been visibly inconsistent at the previous statement boundary. If the system could somehow work out that it is sufficient just to check incoming tuples, then simple update operations would be executed very much more quickly. But such optimizations involve sophisticated expression analysis. While we rightly expect industrial-strength DBMSs to attempt such optimizations, their degree of success is likely to be limited in practice. When we can identify a certain class of constraints that lend themselves to more efficient methods of evaluation, one way of guaranteeing that the system will adopt those more efficient methods is to provide an alternative way of expressing the constraint, applicable only to constraints of that class. If that alternative method is easier for the user to write, and perhaps clearer for the reader too, then the addition to the language can be justified even though it is theoretically redundant.

I will now describe some of the special classes of constraint that have been identified and the shorthands typically used for expressing them, but please note carefully that with just one exception (key constraints) these shorthands are not available in **Tutorial D**. For one thing, they are somewhat controversial but, more importantly, many people have been beguiled by the impoverished state of the existing commercial technology into believing that the term “constraint”, as used in the present context, applies only to what can be expressed using the available shorthands.

Tuple Constraints

The shorthand described in this section is actually frowned upon by many people, the present writer included. I describe it because in most SQL implementations it is the only way of expressing constraints other than key constraints and foreign key constraints.

Consider a constraint whose condition can be expressed as `AND(`*r*`, `*c*`)`, equivalently as `IS_EMPTY(`*r*` WHERE NOT(`*c*`))`, where *r* is a relvar name (i.e., not an invocation of a relational operator) and the condition *c* is an open expression that contains no relvar references. Then *c* can be evaluated against each tuple of *r* without any need to access the database beyond what is needed to obtain the tuples of *r*. Such a constraint is called a *tuple constraint* and the constraint on exam marks expressed in Examples 6.8 and 6.9 is an example.

A typical shorthand for expressing a tuple constraint is to allow the condition *c* to be written inside the definition of the relvar *r* to which it applies. The shorthand is fairly obvious and intuitive. Example 6.10 shows the form it would be likely to take in **Tutorial D** in the extremely unlikely event that the language were ever extended to support the construct.

Example 6.10: Shorthand for a tuple constraint (not allowed in **Tutorial D**)

```
VAR EXAM_MARK BASE RELATION { StudentId SID, CourseId CID,
                              Mark INTEGER }
                          KEY { StudentId, CourseId }
CONSTRAINT Mark_in_range Mark ⩾ 0 AND Mark ⩽ 100 ;
```

As with regular constraint declarations, naming the constraint allows it to be dropped when it is no longer needed.

Now, suppose that not all exams are marked out of 100. Instead, each course has its own maximum mark recorded in the `COURSE` relvar as a value for the attribute `MaxExamMark`. In that case the required constraint, which is set as one of the exercises for this chapter, isn't a tuple constraint because access to another relvar is needed to evaluate it against a given tuple of `EXAM_MARK`. But, as we have already noted, condition *c* specified for a tuple constraint expressed in the manner of Example 6.10, inside the definition of relvar *r*, is equivalent to `IS_EMPTY(`*r*` WHERE NOT(`*c*`))`. It might seem unreasonable if the language allowed some condition to appear as a `WHERE` condition but not as the condition for a constraint declared inside a relvar definition. If *c* is permitted to include explicit relvar references, then the performance advantages to be gained by recognition of tuple constraints (where, by definition, *c* does not include any such references) will accrue only if the DBMS's optimizer is capable of recognizing the cases where *c* does not reference any relvars. That should not be a problem for an industrial strength DBMS. **Tutorial D** does not support this kind of shorthand because the convenience gains are slight and, for teaching purposes at least, it is thought better practice to write such constraints out in full.

Keys

In **Tutorial D** every relvar declaration must include at least one key specification. In Example 6.10 it is `KEY { StudentId, CourseId }`. A key for relvar *r* is a set of attributes of *r* (i.e., a subset of *r*'s heading) such that at no time does *r* contain more than one tuple having any given collection of values for those attributes. Thus, `KEY { StudentId, CourseId }`, included in the relvar declaration for `EXAM_MARK`, specifies a constraint reflecting an integrity rule to the effect that no student can obtain more than one mark for the same exam. Similarly, `KEY { StudentId }`, included in the relvar declaration for `IS_CALLED`, ensures that no student ever has more than one name as far as the database is concerned. The constraint implied by declaration of a key is called, unsurprisingly, a *key constraint*.

To show that `KEY {`*K*`}` included in the declaration for relvar *r* is indeed a shorthand, I note that the constraint could also be expressed as `COUNT(`*r*`) = COUNT(`*r*`{`*K*`})`. If the projection of *r* over the attributes *K* has the same cardinality as *r* itself, then each tuple in the projection has exactly one matching tuple in *r*; otherwise, its cardinality is less than that of *r* itself and at least one tuple in the projection matches more than one tuple in *r*.

Now, the constraint expressed by `KEY {`*K*`}` for relvar *r* captures what is referred to as the *uniqueness* property of a key. A moment's thought reveals that every superset of *K* that is a subset of the heading of *r* must also satisfy this uniqueness property. It is clearly important, when specifying a key, not to include any superfluous attributes. By definition, then, a key has also a property of *irreducibility*: we cannot take any attribute away from a key such that what is left is also a key.

A formal definition for **key** now follows. It uses the term heading to refer to a set of attribute names only, rather than name/type pairs. The projection operator it uses is tuple projection, as defined in Chapter 5, Section 5.10. The definition starts by defining the useful term **superkey**, referring to a subset of the relevant relvar's heading that is a superset of some key for that relvar.

Definitions for superkey and key

Let *K* be a subset of the heading of relvar *r*. Then *K* is **superkey** for *r* if and only if, at all times, if tuples *t1* and *t2* both appear in the body of *r*, and the projection *t1*{*K*} is equal to the projection *t2*{*K*}, then *t1* = *t2* (i.e., they are the same tuple).

K is a **key** for *r* if and only if (a) *K* is a superkey for *r* and (b) no proper subset of *K* is a superkey for *r*.

A superkey satisfies the uniqueness property but not necessarily the irreducibility property. A key satisfies both properties. A **proper superkey** is a superkey that isn't a key.

Note that a relvar can have more than one key. For example, a relvar representing a company's employees might have an attribute `Emp#` representing employee numbers and an attribute `NatIns#` representing national insurance numbers. Each of those attributes on its own satisfies the uniqueness property for keys, so the relvar declaration would include `KEY {Emp#} KEY {NatIns#}`. Don't make the common mistake of writing `KEY {Emp#, NatIns#}` instead! That would mean that employee numbers in combination with national insurance numbers satisfy the uniqueness property, which indeed they do, but that is a weaker constraint, allowing two employees to have the same employee number so long as they have different national insurance numbers (or the same national insurance number so long as they have different employee numbers).

Two special cases of keys are worth noting. The first is where the key is the entire heading, as is the case with our relvar `IS_ENROLLED_ON`. The constraint implied by such a key is such that, were it not enforced, the relvar could at some point in time contain two or more identical tuples—but in that case the value assigned to the relvar would not even be a relation! For that reason the fact that **Tutorial D** requires at least one key to be explicitly specified for each relvar has proved somewhat controversial. Perhaps omission of keys should imply `KEY { ALL BUT }`. An implication of `KEY { ALL BUT }` is that no other key can possibly exist for the relvar it applies to. *Exercise for the reader:* Why is this so?

The second special case is where the key is the empty set. In this case the corresponding key constraint is equivalent to `COUNT(`*r*`) = COUNT(`*r*`{ })`. As projection of *r* over no attributes yields either `TABLE_DEE` (when *r* is not empty) or `TABLE_DUM` (when it is), it follows that `KEY { }` specified for relvar *r* means that *r* can never contain more than one tuple. An implication of `KEY { }` is that no other key can possibly exist for the relvar it applies to. *Exercise for the reader:* Why is this so?

Foreign Keys

A constraint of the form `IS_EMPTY(`*r1* `NOT MATCHING` *r2*`)` is called a *foreign key constraint* if and only if *r1* and *r2* are both relvar references, or relvar references appearing as operands of `RENAME`, and the common attributes of *r1* and *r2* constitute a key of *r2*. In *r1*, the set of common attributes in question is called a *foreign key* and *r1* is the *referencing* relvar of the foreign key; *r2* is the *referenced* relvar. Example 6.3, `IS_EMPTY ( EXAM_MARK NOT MATCHING IS_ENROLLED_ON )`, satisfies those restrictions and is therefore a foreign key constraint: `{StudentId, CourseId}` is a foreign key in the referencing relvar `EXAM_MARK` and the referenced relvar of that foreign key is `IS_ENROLLED_ON`.

Such constraints are very common in practice and their terminology that I have described is very widely used. SQL includes a special shorthand for such constraints. In fact, most SQL implementations *require* this shorthand to be used for expressing foreign key constraints. If **Tutorial D** had a counterpart of SQL's shorthand, then the constraint in Example 6.3 could be expressed, inside the declaration of relvar `EXAM_MARK`, in the following manner:

```
FOREIGN KEY {Student_Id, Course_Id} REFERENCES IS_ENROLLED_ON
```

An invocation of `RENAME` is needed as the referenced "relvar" in the case where the attribute names of the relevant key do not correspond to those of the referencing relvar.

Tutorial D does not support such shorthands, for the following reasons:

1. The requirement for the common attributes to constitute a key of the referenced relvar seems overly restrictive.
2. The requirement for both operands to be simple relvar references (possibly subject to some attribute renaming) also seems overly restrictive.
3. The shorthand isn't much of a shorthand in any case—in the example given it is actually longer than the longhand, as you can see, though the extra length is partly a consequence of having to name the common attributes—you might prefer to write, for example,
   ```
   EXAM_MARK{Student_Id, Course_Id} ⊆
   IS_ENROLLED_ON{Student_Id, Course_Id}
   ```
   for greater clarity.
4. The shorthand is arguably no clearer than the longhand—and might even be thought rather arcane—but, as I have already indicated, the terminology of foreign keys is widely used and understood in the database community. Indeed, I use it myself in the discussion of updating relvars that now follows.

If the restrictions were lifted, then such a shorthand might be considered—but in that case the key words `FOREIGN KEY` would no longer be appropriate.

6.5 Updating Relvars

Recall, from Chapter 1, my view of a database as representing a true account of some enterprise. The verb "to update" is used of databases because it refers to the action of bringing the database up to date in line with changes in the state of the enterprise, when the account the database represents would otherwise be incomplete or untruthful. But the database consists of variables (relvars in the case of relational databases), so to update the database is to update one or more of its variables. In computer languages the general method of updating a variable is called assignment, commonly expressed using the symbol `:=`, as in, for example, `x := x + 1`. The expression on the right-hand side denotes the value that is to become the value of the variable whose name appears on the left-hand side. The value of the expression on the right is termed the *source*, the variable on the left the *target*.

It is normal for assignment to be available in a computer language for variables of all types supported by that language, and **Tutorial D** does indeed allow you to update relvars that way, as illustrated in Example 6.11.

Example 6.11: Enrolling a student on a course using assignment

```
IS_ENROLLED_ON := IS_ENROLLED_ON UNION
                  RELATION { TUPLE { StudentId SID('S3'),
                                     CourseId('C2') } } ;
```

However, although assignment is theoretically sufficient for updating purposes, it is usually more convenient to use a shorthand expressing the difference between the current value of the target relvar and the new value. Sometimes, as in Example 6.11, that difference is just the addition of one or more tuples to the existing set; sometimes it is just changes to some of the attribute values of some of the existing tuples; and sometimes it is just removal of some of the existing tuples. Shorthands for those three particular cases have been referred to as `INSERT`, `UPDATE`, and `DELETE`, respectively, since time immemorial—in other words, even before the advent of relational databases, though of course before that advent the targets of the updates were files, not relvars or SQL tables, and files were collections of records, not sets of tuples.

Do not confuse relational update operators with the read-only operators described in Chapters 4 and 5, which don't update anything. Conversely, update operators do not return values when they are invoked and therefore cannot be used for query purposes.

Descriptions of the **Tutorial D** versions of those update operators now follow.

INSERT

Loosely speaking, `INSERT` adds tuples to a relvar, retaining all the existing tuples. Example 6.12 shows how `INSERT` would be used to the same effect as that of Example 6.11.

Example 6.12: Enrolling a student on a course using INSERT

```
INSERT IS_ENROLLED_ON RELATION { TUPLE { StudentId SID('S3'),
                                         CourseId('C2') } } ;
```

As you can see, "`IS_ENROLLED_ON := IS_ENROLLED_ON UNION`" in Example 6.11 has been replaced by "`INSERT IS_ENROLLED_ON`". We avoid the repeated mention of `IS_ENROLLED_ON` because, as I have already stated, `INSERT` implicitly "retains all the existing tuples", those being the ones contained in the first operand of the `UNION` invocation in Example 6.11.

In general, `INSERT` *rv r*, where *rv* is a relvar name and *r* denotes a relation, is equivalent to *rv* := *rv* `UNION` *r*, implying that *rv* and *r*, now referred to as the target and source, respectively, of the `INSERT` statement, must have identical headings. It is normal practice to require *r* and the current value of *rv* to have no tuples in common (i.e., their bodies to be disjoint), such that the operation fails on an attempt, loosely speaking, to insert a tuple that already exists in the target. The reason for this normal practice lies in the checking of key constraints. Every relvar is subject to at least one key constraint, even when the key in question is the entire heading, as is the case with `IS_ENROLLED_ON`. When processing an `INSERT` statement, the DBMS knows that the current value of the target relvar satisfies all the key constraints, so only the tuples of the source need their key values to be checked for uniqueness. If it encounters a tuple whose key value matches that of an existing tuple or another tuple in the source, the `INSERT` fails and the value of *rv* does not change. Having discovered a clash on key values, most systems do not bother to check to see if in fact the clashing tuples are identical, even when the key value is in fact the entire tuple!

It is easy to imagine, therefore, that Example 6.11, though equivalent to Example 6.12 in its effect, will take very much longer to execute when the current value of `IS_ENROLLED_ON` is of high cardinality. Internally, the DBMS is very likely to execute a relvar assignment by first deleting all the existing tuples, then inserting into the now empty target the tuples of the relation denoted by the expression on the right-hand side of the assignment.

Remarks similar to those on key constraints apply also to other constraints that can be expressed as a condition to be satisfied by all the tuples of a relvar (i.e., by `AND (`*rv,c*`)` where *rv* is a relvar name), including tuple constraints in particular. (Recall that not all such constraints are tuple constraints under the usual definition of that term.) The tuple constraint of Example 6.10 is an obvious case, but consider also foreign key constraints. Suppose we have the constraint condition `IS_EMPTY ( IS_ENROLLED_ON NOT MATCHING IS_CALLED ) AND IS_EMPTY (IS_ENROLLED_ON NOT MATCHING COURSE )`, effectively defining `{ StudentId }` to be a foreign key in `IS_ENROLLED_ON`, referencing `IS_CALLED` and `{ CourseId }` in the same relvar to be a foreign key referencing `COURSE`. Then the DBMS, executing Example 6.12, need only check that student identifier S3 appears in some tuple of the current value of `IS_CALLED` and course identifier C2 appears in some tuple of the current value of `COURSE`. The existing tuples of `IS_ENROLLED_ON` are all guaranteed to satisfy that constraint and therefore do not need to be reexamined. A DBMS faced with the **Tutorial D** method of expressing foreign key constraints might find it quite a challenge to determine that it is only necessary to check the source for the `INSERT`, a simple task for SQL systems that demand such constraints to be expressed using `FOREIGN KEY` syntax.

The source for an `INSERT` can be any expression denoting a relation with heading that of the target. In practice the source is very commonly a relation literal, for the obvious reason that the "new information" being added to the database is indeed new and cannot be derived from the existing value of the database. It is also quite commonly a relation consisting of a single tuple, in which case the syntax for expressing the source for the `INSERT` appears rather heavy-handed. It would not be unreasonable to provide a "single tuple insert" operator in addition to the "multiple tuple insert" of **Tutorial D** but such conveniences can muddy the waters for teaching purposes and we prefer to emphasize the point that a relational DBMS *must* allow more than one tuple to be inserted in a single statement.

Example 6.13 illustrates the convenience of allowing any relation expression to be the source for an `INSERT`. It assumes that all the exam scripts submitted by students have been marked and it has been decided to record marks of zero for students who failed to turn up for an exam they should have sat.

Example 6.13: Awarding zero marks to students who failed to take the exam

```
INSERT EXAM_MARK EXTEND ( IS_ENROLLED_ON NOT MATCHING
                          EXAM_MARK ) : { Mark := 0 };
```

UPDATE

(It is regrettable that the key word `UPDATE` has become so widely accepted as the name of just one particular operator for updating relational databases. Please don't shoot the messenger!)

Loosely speaking, `UPDATE` changes some of the attribute values of some existing tuples of its target relvar. Thus, although some tuples disappear from the target and others arrive in it, so to speak, the cardinality of the relvar does not change. Suppose the exam board for course C2 decides that the exam has been marked too harshly and everybody's mark is to be increased by 5. Example 6.14 shows how.

Example 6.14: Adding 5 to all the marks for course C2

```
UPDATE EXAM_MARK WHERE CourseId = CID('C2') :
                 { Mark := Mark + 5 } ;
```

The syntax is self-explanatory. The `WHERE` specification is optional. It defaults to `WHERE TRUE`, meaning that the specified changes are to be applied to all existing tuples in the target relvar. The expression `Mark := Mark + 5` is an *attribute assign*. When several attribute assigns are needed they are separated by commas.

As with INSERT, various optimizations are available to the DBMS when it comes to checking constraints. For example, only the tuples satisfying the WHERE condition need to be examined, after application of the attribute assigns, to see if they satisfy those constraints that can be checked "a tuple at a time", as discussed in the section on INSERT. Moreover, of those constraints the DBMS need only consider the ones that involve attributes whose names appear as attribute assign targets in the UPDATE statement. In Example 6.14 no key or foreign key constraints need to be checked, because Mark is the only attribute assign target and that attribute does not appear in the only key for EXAM_MARK, nor is it a common attribute for any foreign key constraint. But Mark *is* involved in the constraint to ensure that all marks are in the range 0 to 100. If any student has already scored 96 or more in the exam for course C2, the exam board might have to revise its upgrade policy for that exam!

That an UPDATE invocation is indeed shorthand for some assignment is illustrated by Example 6.15, which is equivalent to Example 6.14.

Example 6.15: Adding 5 to all the marks for course C2, the hard way

```
EXAM_MARK := EXAM_MARK WHERE NOT ( CourseId = CID('C2') )
             UNION
             EXTEND ( ( EXAM_MARK WHERE CourseId = CID('C2') )
                      RENAME { Mark AS Xmark } ) :
                    { Mark := Xmark + 5 } { ALL BUT Xmark } ;
```

You can easily see that, compared with INSERT, UPDATE offers quite generous shorthands! Note how the first operand of the UNION invocation preserves, so to speak, the unaffected tuples of EXAM_MARK.

DELETE

Loosely speaking, DELETE removes some existing tuples from its target relvar. Suppose the university decides that course C3 is to be withdrawn. Example 6.16 shows how.

Example 6.16: Withdrawing course C3, using DELETE

```
DELETE COURSE WHERE CourseId = CID('C3') ;
```

Still speaking loosely, every tuple that satisfies the given WHERE condition is deleted and tuples that do not satisfy it remain.

Now, it might seem that, as the only tuples remaining in the target are ones that are already known to satisfy all constraints on the target that can be checked "a tuple at a time", there is no need for the DBMS to check such constraints at all when executing a DELETE statement. But we have been assuming that { CourseId } in IS_ENROLLED_ON is a foreign key referencing COURSE. If any students are recorded as being currently enrolled on course C3, then the DELETE statement in Example 6.16 must fail, for then the result of IS_ENROLLED_ON NOT MATCHING COURSE would not be empty as required.

Tutorial D also supports a second form of DELETE, where the tuples to be deleted are specified by a relation expression. For example, suppose the INSERT shown in Example 6.12 turned out to be a mistake by an end-user of *Rel*'s Dbrowser. Then the user could simply recall the INSERT statement and replace the word INSERT by DELETE:

```
DELETE IS_ENROLLED_ON RELATION { TUPLE { StudentId SID('S3'),
                                         CourseId('C2') } } ;
```

So Example 6.16 is actually shorthand for

```
DELETE COURSE COURSE WHERE CourseId = CID('C3') ;
```

!

Of course, if there *are* any students recorded as enrolled on a course that is being withdrawn, then those records are surely obsolete. To update the database to reflect the real world change and also ensure that it satisfies the foreign key constraint involving `IS_ENROLLED_ON` and `COURSE`, we clearly need to delete all the enrolments on C3 as well as deleting the course itself. We can do that by taking care to delete the enrolments first (a course is allowed to exist with no enrolments) and *then* delete the course. But there's a better solution, avoiding the need for care over the order of events, and having other advantages too. It is called *multiple assignment.*

Multiple Assignment

Example 6.17 shows how to delete course C3 and all its current enrolments at a single stroke.

Example 6.17: Withdrawing course C3 and deleting any enrolments on C3

```
DELETE COURSE WHERE CourseId = CID('C3') ,
DELETE IS_ENROLLED_ON WHERE CourseId = CID('C3') ;
```

It might appear at first glance to consist of two `DELETE` statements, the first of which you would expect to fail by violating a (foreign key) constraint. On closer inspection you should notice that it is in fact a single statement, there being only one semicolon. The first invocation of `DELETE` ends in a comma, not a semicolon. Now recall that only semicolons denote statement boundaries, and statement boundaries are the points at which the database is required to be consistent with the database constraint. It is not required to be consistent at a point indicated by a mere comma—for at such a point any inconsistency arising would be "visible" only to the DBMS and not to any user.

Separating invocations of update operators—called *assigns*—by commas to form a single statement is **Tutorial D**'s method of expressing multiple assignment. It is important to realize that the individual assigns are considered to be executed concurrently, in parallel, regardless of the order in which they are written. This implies that all sources must be evaluated before any assigns are executed. Example 6.18, containing two statements but three assigns, illustrates this point using simple assigns to integer variables.

Example 6.18: A consequence of simultaneity

```
X := 1;

X := X + 1,
Y := X + 1 ;
```

The first statement assigns 1 to the variable `X`. The result of the second statement is then to assign 2 to both variables, `X` and `Y`. If we replace the comma by a semicolon, then instead the result of executing what now becomes three statements would be to assign 2 to `X` and 3 to `Y`.

Multiple assignment is more than a mere convenience in certain circumstances. Sometimes a problem arises to which it is the only solution, as shown in the following scenario.

Consider the business of taking purchase orders and delivering the purchased items to customers. Each purchase item belongs to a particular delivery and each delivery consists of one or more items. Deliveries are identified by reference numbers. A delivery reference number is attached to each item to identify the delivery it belongs to. The delivery reference number of an item must be that of a known planned or completed delivery and every delivery must contain at least one item—consider the consequences if either of those constraints was not in force. So we have the constraint `IS_EMPTY ( DELIVERY NOT MATCHING DELIVERY_ITEM ) AND IS_EMPTY ( DELIVERY_ITEM NOT MATCHING DELIVERY )`. Without multiple assignment we cannot insert a tuple *t* into `DELIVERY` unless some tuple *tt*, matching *t*, exists in `DELIVERY_ITEM`, but tuple *tt* cannot possibly exist in `DELIVERY_ITEM` because if it did it would violate the condition `IS_EMPTY ( DELIVERY_ITEM NOT MATCHING DELIVERY )`. With multiple assignment we can resolve the apparent impasse by inserting into both relvars simultaneously. Similarly, if a delivery is cancelled we need to delete all the relevant tuples from both `DELIVERY` and `DELIVERY_ITEM` simultaneously.

Transactions

Although multiple assignment is the recommended method of ensuring a database's consistency and completeness, commercial DBMSs at the time of writing do not support it. Instead, they allow several update statements to be batched together to form a *transaction,* whereby the effects of each individual update statement are visible to the user who submits them to the DBMS, but are not visible to other users until the transaction is *committed*—if indeed it is ever committed, for the user also has the option to cancel the transaction, thus undoing all the updates submitted up to the point of cancellation. Typically, the database is permitted to be inconsistent with its declared constraints until the transaction is committed. In SQL, for example, the user may specify that the checking of certain specified constraints be *deferred* until either (a) that deferment request is cancelled or (b) the transaction is committed.

Tutorial D supports transactions but does not support the deferring of constraint checking. Thus, developers of applications and user-defined operators are able to use program code that assumes that the database is consistent whenever that code is executed. Transactions are now just a convenience, whereby the database, though always consistent, might sometimes be incomplete—though this incomplete state is visible only to the user owning the transaction. The following statements constitute **Tutorial D**'s support for transactions:

- **BEGIN TRANSACTION** is self-explanatory, but note that transactions can be "nested"—a transaction can be started within an existing transaction.
- **COMMIT** commits the updates of the most recently started transaction and ends that transaction, but those updates become visible to other users only if the transaction in question is an outermost one (i.e., not nested inside another transaction).
- **ROLLBACK** cancels the updates of the most recently started transaction and ends that transaction. The cancelled updates include any resulting from some nested transaction that has been committed.

I have now described everything needed for definition, manipulation, and maintaining the integrity of a relational database. Just one more topic needs to be addressed to complete the account of the foundational theory for such databases: database design.

EXERCISES

1. (Repeated from the body of the chapter).
   a) An implication of `KEY { ALL BUT }` is that no other key can possibly exist for the relvar it applies to. Why is this so?
   b) An implication of `KEY { }` is that no other key can possibly exist for the relvar it applies to. Why is this so?

2. Suppose the relvar definition for `COURSE` is extended to include an attribute `MaxExamMark`, whose value in each tuple is the maximum mark obtainable for that course's exam. `{StudentId, CourseId}` is a foreign key in `EXAM_MARK`, referencing `IS_ENROLLED_ON`. A constraint is needed to ensure that no student is awarded a mark greater than the relevant maximum.
   a) Write a **Tutorial D** `CONSTRAINT` statement to address this requirement, where the constraint condition is an invocation of `IS_EMPTY`.
   b) Complete the following statement to make it equivalent to the one you wrote for part (a):
   ```
   CONSTRAINT ... AND(EXAM_MARK, ... ) ;
   ```

3. Now suppose that instead of there being a recorded maximum mark of each exam the maximum score for each question in each exam is recorded in the following relvar:
   ```
   VAR EXAM_QUESTION BASE RELATION { CourseId CID,
          Question# INTEGER, MaxMark INTEGER }
          KEY { CourseId, Question# } ;
   ```
   For each course, the exam questions are supposed to be numbered sequentially, starting at 1.
   a) Write a **Tutorial D** `CONSTRAINT` statement to address this requirement.
   b) Suppose the questions are subdivided into parts, a, b, c and so on, up to a maximum of six parts, and maximum marks are given for each part rather than for each question. Again, the parts for each question must be "numbered" sequentially, starting at a. Write a **Tutorial D** `CONSTRAINT` statement to address *this* requirement.
   c) Devise shorthands, in the style of **Tutorial D**, for expressing constraints of the kinds found in your solutions to a. and b.

4. Using *Rel*, with the suppliers-and-parts database set up for the *Rel* exercises given at the end of Chapter 4, write **Tutorial D** integrity constraints to express the following requirements:
   a) Every shipment tuple must have a supplier number matching that of some supplier tuple.
   b) Every shipment tuple must have a part number matching that of some part tuple.
   c) All London suppliers must have status 20.
   d) No two suppliers can be located in the same city.
   e) At most one supplier can be located in Athens at any one time.
   f) There must exist at least one London supplier.
   g) The average supplier status must be at least 10.
   h) Every London supplier must be capable of supplying part P2.

7 Database Design I: Projection-Join Normalization

7.1 Introduction

Relational database design takes a statement of requirements and produces a database definition to address those requirements. The definition consists of a collection of relvar and constraint definitions. As Chris Date puts it in [9] under the heading **logical database design**:

> Ideally, the goal is to produce a design that's independent of all considerations having to do with either physical implementation or specific applications—the latter objective being desirable for the good reason that it's generally not the case that all uses to which the database will be put are known at design time.

The production and format of a precise and complete requirements statement are beyond the scope of this book. Suffice it here just to say that the statement usually takes the form of a collection of "business rules" and/or some kind of "entity/relationship model" presented in some agreed notation. Business rules are expressed in this chapter for some examples, in an intuitive and somewhat informal style thought to be good enough for the purpose at hand. The fact is, though, that even when the requirements are 100% clear there are usually some design choices to be made: in other words, there can be several significantly different designs to implement any given requirement statement.

What common kinds of alternative might the designer encounter and in each case what considerations should guide the designer in arriving at the preferred choice? In this book I describe and discuss several common alternatives under the headings **Projection-join Normalization** (this chapter), **Group-Ungroup and Wrap-Unwrap Normalization**, **Restriction-Union Normalization**, **Surrogate Keys**, and **Representing "Entity Subtypes"**. The reader should be warned, though, that relational database theory has very little science to offer regarding database design, and what little science it does offer is almost entirely within the first of these topics, projection-join normalization, to which the rest of this chapter is devoted. For the others, described in Chapter 8, we can do no more than make note of the choices and suggest some guidelines.

7.2 Avoiding Redundancy

A common issue in database design concerns *redundancy*—recording the same information more than once. For example, redundancy is exhibited in this book's very first example of a relation: Figure 1.2 in Chapter 1, where the information that student S1's name is Anne is recorded twice. The explanation accompanying this figure indicates that the relation is the current value of a relvar named `ENROLMENT`, so we can safely conclude that the possibility of redundancy is a consequence of the database design—a student's name is recorded as many times as that student has concurrent enrolments.

As a rule of thumb we assume that redundancy is normally to be avoided. Intuitively it seems untidy and possibly inefficient to record the same fact more than once. Stating S1's name repeatedly for each course she is enrolled on obviously entails more work than stating it just once. What's more, we have to be certain she is given the same name, spelled the same way, every time, and making sure of that entails more work too. In Chapter 4, Figure 4.1 shows an alternative design involving the two relvars, `IS_CALLED` and `IS_ENROLLED_ON`. In that preferred design the redundant repetitions of students' names are avoided: each name is recorded just once, in `IS_CALLED`. In the same chapter, Section **4.4 `JOIN` and `AND`**, I showed that the relation denoted by the expression `IS_CALLED JOIN IS_ENROLLED_ON` is the very one shown in Figure 1.2. Later in the same chapter, Section **4.6 Projection and Existential Quantification**, Example 4.5, I showed how we can use projection to decompose `ENROLMENT` into those two relvars, `IS_CALLED` and `IS_ENROLLED_ON`. (Please ignore, for the moment, the fact that Figure 4.1 actually includes some information—that student S5 is called Boris—not represented in Figure 1.2, rendering the designs not exactly equivalent. I'll come back to this point later.) Each projection gives us (a) the heading for one of those relvars, and (b) the initial value for the same relvar. So, we have two designs—let's call them Design A and Design B—that are equivalent in the following sense: if Design A (single relvar) is chosen, then the current value for Design B can always be obtained using projection, whereas if Design B is chosen, then the current value for Design A can always be obtained using `JOIN`. *Projection-join normalization* concerns design choices of that particular kind.

When something can be expressed in two or more equivalent ways, we sometimes have reason to prefer just one of those ways. For example, the fraction expressing "two thirds" is normally written as ⅔ in preference to, say, 4/6. The term *normal form* refers to such preferences (*canonical form* means the same thing, but normal form is the term conventionally used in the relational database context).

Various normal forms have been defined for relvars. When a proposed relational database design includes a relvar that does not satisfy a certain normal form, *normalization* is the process by which that relvar can be replaced by one or more different relvars that do satisfy that normal form and still meet the original requirements. Projection-join normalization is a process whereby normal forms are obtained using projections, such that joins can be used to reverse the process. We need to understand in what circumstances we can indeed use projections for this purpose, in what circumstances it is advantageous to do so, and in what circumstances it might be better *not* to do so. We turn to these issues in the next section.

7.3 Join Dependencies

A given relvar *r* can be decomposed into two or more relvars to yield an equivalent design only when *r* is subject to a special kind of constraint called a *join dependency.* Figure 7.1 depicts the current value of a relvar, `WIFE_OF_HENRY_VIII` that is subject to such a constraint. (Students of English history during the period of the Tudor dynasty, 1485–1603, are traditionally taught the mnemonic, "divorced, beheaded, died, divorced, beheaded, survived" by which to remember what became of each of King Henry VIII's six wives.)

WIFE_OF_HENRY_VIII

Wife#	FirstName	LastName	Fate
1	Catherine	of Aragon	divorced
2	Anne	Boleyn	beheaded
3	Jane	Seymour	died
4	Anne	of Cleves	divorced
5	Catherine	Howard	beheaded
6	Catherine	Parr	survived

Figure 7.1: Example to illustrate join dependency

Note first of all that this design does *not* exhibit redundancy. The example is given merely to introduce you to the concept of join dependency, using a simple case to illustrate it. The predicate for `WIFE_OF_HENRY_VIII` is "The first name of Henry VIII's wife number *Wife#* is *FirstName* and her last name is *LastName* and *Fate* is what became of her." The appearances of the word "and" in this predicate indicate that it is "decomposable" into two or more simpler predicates. For example:

1. "The first name of Henry VIII's wife number *Wife#* is *FirstName*."
2. "The last name of Henry VIII's wife number *Wife#* is *LastName* and *Fate* is what became of her."

The relations corresponding to predicates 1 and 2 are shown, in Figure 7.2, as the current values of relvars `W_FN` (wives' first names) and `W_LN_F` (wives' last names and fates), respectively.

W_FN

Wife#	FirstName
1	Catherine
2	Anne
3	Jane
4	Anne
5	Catherine
6	Catherine

W_LN_F

Wife#	LastName	Fate
1	of Aragon	divorced
2	Boleyn	beheaded
3	Seymour	died
4	of Cleves	divorced
5	Howard	beheaded
6	Parr	survived

Figure 7.2: A decomposition of WIFE_OF_HENRY_VIII

Note that

- `W_FN = WIFE_OF_HENRY_VIII {Wife#, FirstName}`
- `W_LN_F = WIFE_OF_HENRY_VIII {Wife#, LastName, Fate}`
- `WIFE_OF_HENRY_VIII = W_FN JOIN W_LN_F`

The constraint determining that `WIFE_OF_HENRY_VIII` can be decomposed and subsequently recomposed in these ways is a join dependency. The join dependency can be defined as shown in Example 7.1,

Example 7.1: A constraint condition expressing a join dependency

```
WIFE_OF_HENRY_VIII = WIFE_OF_HENRY_VIII {Wife#, FirstName}
                     JOIN
                     WIFE_OF_HENRY_VIII {Wife#, LastName, Fate}
```

but, as we shall see, that constraint is actually implied by the `KEY` specification given in the relvar definition for `WIFE_OF_HENRY_VIII` and so does not need to be spelled out again. A join dependency, commonly abbreviated JD, is any condition that can be expressed in this form, denoting that a given relvar is at all times equal in value to the join of two or more projections of its current value. Using the conventional notation for join dependencies we can write the one in Example 7.1 as follows:

```
* { { Wife#, FirstName }, { Wife#, LastName, Fate } }
```

For convenience, I shall refer to the components of a JD—{ `Wife#`, `FirstName` } and { `Wife#`, `LastName, Fate` } in this example—as *projections*, and the number of projections as the *degree* of the JD. This one is of degree 2 and is therefore called a binary JD. The symbol * is often used in textbooks for the join operator. Because the operands of a JD are projections they are indicated by lists of attribute names enclosed in braces; those operands are in turn enclosed in braces because in general a join dependency can involve any number of projections and, thanks to the commutativity and associativity of the join operator, the order in which those projections are written is insignificant.

Note that the formulation shown for one of the JDs to which `WIFE_OF_HENRY_VIII` is subject makes no mention of that relvar. Whenever we mention a JD it must be clear from the context to which relvar it applies. We normally do that by stating whether the given JD *holds* in that relvar. The JD in our example does indeed hold in `WIFE_OF_HENRY_VIII`. By contrast, the JD

```
* { { Wife#, FirstName }, { LastName, Fate } }
```

for example, does *not* hold in that relvar, because the following tuple, among several others, appears in the join of those two projections but does not appear in the current value of `WIFE_OF_HENRY_VIII`:

```
TUPLE { Wife# 2, FirstName 'Anne',
        LastName 'Parr', Fate 'survived' }
```

Note that in the projection over { `LastName`, `Fate` } we lose the information that Seymour is the last name of wife number 3, for example. Conversely, in the join we "gain" the *misinformation* represented by those tuples that do not appear in `WIFE_OF_HENRY_VIII`. To put it more accurately, the predicate for `WIFE_OF_HENRY_VIII` does not apply to the result of this join, so the information represented is different too. (*Exercise for the reader:* what predicate does apply to it?)

Note that a JD cannot possibly hold in its applicable relvar, *r*, unless each attribute of *r* appears in at least one of the projections. If that is not the case, then the join of those projections does not have the same heading as *r* and therefore cannot be equal to *r*.

You have probably noticed that `W_LN_F`, in whose predicate the word "and" still appears, can be further decomposed. I'll come back to this point in a moment. `W_FN`, however, cannot be further decomposed. We say that relvar W_FN is an *irreducible relvar.* We also say that relvar `W_FN` is in *sixth normal form* (6NF), whereas `W_LN_F` is not in 6NF and nor is `WIFE_OF_HENRY_VIII`. You are right in guessing from the name, sixth normal form, that other normal forms have been identified, at least five of which are identified by numbers. In fact several others have been defined around the idea of eliminating certain JDs, varying according to exactly which particular kinds of JD they eliminate. Fortunately, some of them can now safely be regarded as preliminary ideas by researchers, later subsumed by more general and more useful definitions. Nowadays it is sufficient to study just three "JD-eliminating" normal forms. In this book I refer to them as projection-join normal forms, but please note that the term *projection-join normal form* (PJ/NF) is used by some writers—including its originator, Fagin [13]—to refer specifically to just one of these three (namely, 5NF).

As it happens, 6NF is the strongest projection-join normal form that can be defined, because the only JDs that can hold in a 6NF relvar are ones that cannot be eliminated at all. 6NF is also perhaps the easiest to understand because the class of JDs eliminated by it is simple to define, as I will now show.

Trivial and Nontrivial JDs

6NF doesn't eliminate all JDs. Take `W_FN`, for example. The following JDs, among others, all hold in `W_FN`:

- `* { { Wife#, FirstName }, { Wife# } }`
- `* { { Wife#, FirstName }, { FirstName } }`
- `* { { Wife#, FirstName }, { } }`
- `* { { Wife#, FirstName } }`

That last one is a unary JD. Exactly one unary JD holds in every relvar, namely the one whose projection includes all the attributes. It is of no importance, of course. We admit it as a JD simply to avoid a needless complication in the definition of that term.

In each of the JDs shown above, one of the projections is the identity projection (i.e., over all the attributes of the applicable relvar). A moment's thought should convince you that a JD involving the identity projection cannot fail to hold in its applicable relvar. To spell it out, though, in general, if *r2* is a projection of *r1*, then *r1* `JOIN` *r2* is equal to *r1;* for every tuple of *r2* is by definition a subset of some tuple of *r1* and therefore matches at least one tuple of *r1*. Moreover, every tuple *t2* of *r2*, when joined with a tuple *t1* of *r1,* yields that tuple *t1*. We say, therefore, that a JD that includes the identity projection of its applicable relvar is *trivial.* The classification of JDs into trivial and nontrivial ones enables us easily to define 6NF:

Relvar *r* is in **sixth normal form** (6NF) if and only if every join dependency that holds in *r* is trivial.

We will now complete the decomposition of `WIFE_OF_HENRY_VIII` into relvars that are all in 6NF. Then we will be able to think about the relative advantages and disadvantages of the two designs.

Further Decomposition of `WIFE_OF_HENRY_VIII`

The JD `* { { Wife#, LastName }, { Wife#, Fate } }` holds in `W_LN_F` and is nontrivial. `W_LN_F` can therefore be decomposed as shown in Figure 7.3.

W_LN

Wife#	LastName
1	of Aragon
2	Boleyn
3	Seymour
4	of Cleves
5	Howard
6	Parr

W_F

Wife#	Fate
1	divorced
2	beheaded
3	died
4	divorced
5	beheaded
6	survived

Figure 7.3: Further decomposition of WIFE_OF_HENRY_VIII

Although I decomposed WIFE_OF_HENRY_VIII in two stages, I could of course have done it in a single step. Had I done so, I might have been looking at the following single JD—a ternary JD (having three projections) that holds in WIFE_OF_HENRY_VIII—in place of the two that governed my previous two stages:

```
* { { Wife#, FirstName }, { Wife#, LastName },{ Wife#, Fate } }
```

Now W_LN and W_F are both, like W_FN, in 6NF. By contrast, WIFE_OF_HENRY_VIII is not in 6NF. Might it be better to decompose it into those 6NF relvars? We must examine the two designs in detail to assess the situation and come to a decision.

Assessment of 6NF decomposition

So far we have shown only the structural aspects of two equivalent designs to be considered. Design A is the single relvar design; Design B uses the three 6NF relvars, W_FN, and W_LN, and W_F.

Here are the relvar definitions for the two designs:

Design A

```
VAR WIFE_OF_HENRY_VIII BASE RELATION { Wife# INTEGER,
                                       FirstName CHAR,
                                       LastName CHAR,
                                       Fate CHAR }
                                  KEY { Wife# } ;
```

Design B

```
VAR W_FN BASE RELATION { Wife# INTEGER,
                         FirstName CHAR }
                    KEY { Wife# } ;

VAR W_LN BASE RELATION { Wife# INTEGER,
                         LastName CHAR }
                    KEY { Wife# } ;

VAR W_F BASE RELATION { Wife# INTEGER,
                        Fate CHAR }
                   KEY { Wife# } ;
```

To assess these alternative designs we need to consider also the constraints that must be applied to those relvars to complete the designs. Assuming that Design A is correct, we can infer some of the requirements, which I express as the following business rules:

BR1: Every wife has a wife number.
BR2: No two distinct wives have the same wife number.
BR3: Every wife has a first name.
BR4: Every wife has a last name.
BR5: Every wife has a fate.

BR1, BR3, BR4, and BR5 are implied by the very structure of relvar `WIFE_OF_HENRY_VIII`, because every tuple in the body of a relation has exactly one value for every attribute of that relation, by definition. BR2 is implied by the specification `KEY { Wife# }`. But these business rules are not reflected in the relvar definitions of Design B, apart from BR1. As things stand, the keys defined for the three relvars prevent any wife from having more than one first name, last name, or fate, but it is possible for a wife to have a wife number but no first name, or no last name, or no fate. So the "exactly one" stipulations of BR3, BR4, and BR5 are not met. As for BR2, all we can say is that no two wives with first names have the same wife number, nor do any two wives with last names, nor do any two wives with fates. To enforce the given business rules we need a constraint to the effect that every wife number appearing in any one of those three relvars appears in the other two as well. In **Tutorial D** we could express that as shown in Example 7.2.

Example 7.2: Enforcing BRs 1 to 5 in Design B

```
CONSTRAINT BRs_1_to_5
     W_FN { Wife# } = W_LN { Wife# } AND
     W_LN { Wife# } = W_F { Wife# } AND
     W_FN { Wife# } = W_F { Wife# } ;
```

(Any one of those three conjuncts can of course be omitted as implied by the other two.) It is this constraint—however it is expressed—that makes Design B significantly more complex than Design A. (It gives rise to a performance challenge for the DBMS, too. That could perhaps be addressed by provision of some suitable shorthand but note that in SQL systems, which process foreign key constraints efficiently, we would need no less than six foreign key definitions, two for each relvar.) There are also unpleasant implications for certain kinds of update in Design B. To "insert a new wife" or "delete a wife" we need multiple assignment on all three relvars. That's not available in existing commercial DBMSs (in 2009) but the alternative solutions typically provided are no less complicated in their implementation. (Foreign keys are discussed in Chapter 6, Section 6.4, under the heading **Foreign Keys**. Multiple assignment is described in Section 6.5 of that chapter, under the heading **Multiple Assignment**.)

It seems there is little to commend Design B for the particular example at hand, apart, perhaps, from the fact that an update affecting just one of the three relvars will not interfere with other users' access to the other two while that update is in process.

Now, the foregoing analysis assumes that Design A and Design B are both correct. But imagine the database existing during the lifetime of King Henry VIII, the designer having chosen Design A. Anne Boleyn has lost her head and the king has just married Jane Seymour, which wife must now be added to the database. What value is to be given for the attribute `Fate`? 'To be determined', perhaps? Or the empty character string? In either case, the predicate I gave for `WIFE_OF_HENRY_VIII`, including the words "and *Fate* is what happened to her", is no longer applicable. Nor is business rule BR5.

Perhaps a separate relvar for recording the wives' fates, *where known*, would be a good idea after all, allowing us to record Jane Seymour's wife number, first name, and last name without recording anything regarding the fate of her marriage. Suddenly I am touching on one of the most difficult and controversial issues surrounding relational database theory and practice: the so-called problem of "missing information", addressed in SQL by the introduction of an innocent-looking little thing that actually undermines the very foundations of relational theory by allowing this thing—called `NULL`—to appear in place of an attribute value (`NULL` is not a value and consequently gives rise to some very strange phenomena when it appears in place of a value). Further discussion of `NULL` is beyond the scope of the present book but is dealt with in detail in its companion book *SQL: A Comparative Survey*. Here we assume that a correct design obviates the need to worry about "what to put when there is nothing to put" by not requiring anything to be put! But even then there are choices to consider and these are discussed in Chapter 8, Section 8.4, **Representing "Entity Subtypes"**.

Returning to the subject of projection-join normalization, I have explained 6NF and noted that relvar WIFE_OF_HENRY_VIII is not in 6NF but, assuming it is a correct design for the given requirements, is very likely to be preferred to a 6NF design. Although certain nontrivial join dependencies hold in it, it exhibits no redundancy. By contrast, ENROLMENT does exhibit redundancy as already noted (Anne's name is recorded more than once) and you have already seen how decomposition, into IS_CALLED with attributes StudentId and Name and IS_ENROLLED_ON with attributes StudentId and CourseId, addresses that problem. Because this decomposition was available to us, we can conclude that the two projections by which it was achieved constitute a JD that holds in ENROLMENT:

```
* { { StudentId, Name }, { StudentId, CourseId } }
```

If decomposition is desirable for ENROLMENT but not for WIFE_OF_HENRY_VIII, then there must be some difference in kind between the JD that holds in ENROLMENT and those that hold in WIFE_OF_HENRY_VIII. It is this difference that allows us to define a projection-join normal form that is not as strong as 6NF but is much more desirable, in general, than 6NF. You won't be surprised to hear that it is called *fifth normal form* (5NF).

7.4 Fifth Normal Form

WIFE_OF_HENRY_VIII is in 5NF. So are IS_CALLED and IS_ENROLLED_ON, but ENROLMENT is not. What is special about that ternary JD that holds in WIFE_OF_HENRY_VIII? Here it is again, with its distinguishing feature shown in bold:

* { { **Wife#**, FirstName }, { **Wife#**, LastName }, { **Wife#**, Fate} }

There are two significant points to be made about the attribute Wife#:

a) It appears in each projection of the JD.
b) { Wife# } is a key of WIFE_OF_HENRY_VIII (in fact the only key).

By contrast, consider * { { StudentId, Name }, { StudentId, CourseId } }, the JD holding in ENROLMENT. Although its projections have an attribute, StudentId, in common, that common attribute does not constitute a key of ENROLMENT, and that is what gives rise to the redundancy in this particular case. Although student identifier S1 is always paired with the same name, Anne, that pairing can appear in several different tuples in the current value of ENROLMENT. By contrast, the pairing of a particular wife number with a particular first name, last name, or fate, cannot possibly appear in several different tuples of WIFE_OF_HENRY_VIII, because it is not even possible for the same wife number to appear in more than one tuple, thanks to the constraint implied by the specification KEY { Wife# }. These observations, among others, lead us to a definition for fifth normal form.

Relvar *r* is in **fifth normal form** (5NF) if and only if every join dependency that holds in *r* is implied by the keys of *r*.

`WIFE_OF_HENRY_VIII` satisfies this definition because every JD that holds in it is one in which every projection either includes its only key, {`Wife#`}, or is redundant. (A projection is redundant if all of its attributes are included in one of the other projections. For example, every trivial JD contains at least one redundant projection. You can verify for yourself that if a redundant projection is removed from a JD that holds, then the resulting JD also holds.) In other words, given the complete relvar definition for `WIFE_OF_HENRY_VIII` in **Tutorial D**, thus knowing only its attributes and its single key, we can write down every JD that holds in it:

- `*{{Wife#, FirstName},{Wife#,LastName},{Wife#,Fate}}`
- `*{{Wife#, FirstName, LastName},{Wife#,Fate}}`
- `*{{Wife#, FirstName, Fate },{Wife#,LastName},{Fate}}`
- … and so on

(The third one listed above includes a redundant projection, {`Fate`}.)

For convenience in the discussion that follows, I use the following terms:

- *rogue JD* for a JD that does not satisfy the condition given in the foregoing definition of 5NF
- *5NF relvar* for a relvar that is in 5NF
- *non-5NF relvar* for a relvar that is not in 5NF

Now, a 5NF relvar is guaranteed never to exhibit redundancy of the kind that can be eliminated by projection-join normalization. Contrary to what I wrote here in the first edition of this book, non-5NF relvars do exist that are redundancy-free (i.e., do not exhibit redundancy), but the discovery of such relvars didn't come about until after 2009 when the first edition was written. For present purposes, let us assume, albeit falsely, that 5NF is both sufficient and necessary for avoiding such redundancy. It is generally held that we should aim for designs consisting exclusively of 5NF relvars, because of an overarching need to avoid redundancy. In any case, if there appear to be good reasons for not going to that extreme, the designer should be well aware of the potential costs involved in violating 5NF. Let us therefore have a closer look at the non-5NF relvar ENROLMENT to discover the problems caused by the redundancy it exhibits. Here is a relvar definition for it:

```
VAR ENROLMENT BASE RELATION { StudentId SID,
                              Name NAME,
                              CourseId CID }
                         KEY { StudentId, CourseId } ;
```

We can infer from this definition that a student, enrolled on a course, has exactly one name *in connection with that enrolment*. But actually there is a business rule to the effect that every student has exactly one name, regardless of enrolments. Student S1 is always called Anne. That rule is exactly what the JD * { { StudentId, Name }, { StudentId, CourseId } } really means, and we cannot infer that JD from the relvar definition—the attributes and key—alone. To show that we can infer it from the given business rules, consider what happens if S1's name is recorded as Anne for course C1 but Ann (without the "e") for course C2. In that case the tuples TUPLE { StudentId SID('S1'), Name NAME('Anne') } and TUPLE { StudentId SID('S1'), Name NAME('Ann') } both appear in the projection ENROLMENT{StudentId, Name} and therefore the following tuples both appear in the join of that projection with ENROLMENT{StudentId, CourseId}:

```
TUPLE { StudentId SID('S1'), Name NAME('Anne'),
        CourseId CID('C1') }

TUPLE { StudentId SID('S1'), Name NAME('Ann'),
        CourseId CID('C1') }
```

But the second of those does not appear in ENROLMENT. Therefore the constraint defined by that JD does not hold after all and the business rule it would express is violated. To enforce that business rule we need to define an appropriate constraint. Following Example 7.1 we could write, in **Tutorial D**,

```
CONSTRAINT JD_in_ENROLMENT
  ENROLMENT = JOIN { ENROLMENT {StudentId, Name},
                     ENROLMENT {StudentId, CourseId} } ;
```

or, less directly,

```
CONSTRAINT JD_in_ENROLMENT
  COUNT (ENROLMENT {StudentId, Name} =
  COUNT (ENROLMENT {StudentId} ;
```

meaning that there are as many distinct <StudentId, Name> pairs appearing in ENROLMENT as there are distinct StudentId values, which implies that no StudentId value is paired with more than one Name value.

Now, look at the database design for the decomposition into IS_CALLED and IS_ENROLLED_ON. First, its structural part:

```
VAR IS_CALLED BASE RELATION { StudentId SID,
                              Name NAME }
                        KEY { StudentId } ;

VAR IS_ENROLLED_ON BASE RELATION { StudentId SID,
                                   CourseId CID }
                             KEY { StudentId, CourseId } ;
```

We must check the business rules that (presumably) led to the single relvar design, with its constraint JD_in_ENROLMENT, to determine what additional constraints need to be included in the new design. Here are those business rules:

BR1: An enrolment is uniquely identified by a student identifier and a course identifier.
BR2: A student enrolled on a course has exactly one name for that enrolment.
BR3: All enrolments for the same student have the same name for that student.
BR4: Every student whose name is recorded is enrolled on at least one course.

Now, how does our proposed new design address those business rules?

- BR1 is implied by `KEY { StudentId, CourseId }` in the declaration for `IS_ENROLLED_ON` (and is in any case implied by the heading of that relvar, there being no non-key attributes).
- For BR2 we need an additional constraint to ensure that at no time does any tuple in `IS_ENROLLED_ON` fail to match some tuple in `IS_CALLED`. That's a foreign key constraint on `StudentId` in `IS_ENROLLED_ON`, which we might express in **Tutorial D** as

```
      CONSTRAINT Student_must_have_a_name
         IS_EMPTY ( IS_ENROLLED_ON NOT MATCHING IS_CALLED ) ;
```

- For BR3 we need to consider the join of `IS_CALLED` and `IS_ENROLLED_ON`. Thanks to the key constraint expressed by `KEY { StudentId }` in the declaration for `IS_CALLED`, that join is always "many-to-one", as opposed to "many-to-many", because no tuple in `IS_ENROLLED_ON` can possibly match more than one tuple in `IS_CALLED`. So the new design does already enforce BR3.
- For BR4 we need a constraint similar to that for BR2 but "in the other direction", so to speak, to ensure that at no time does any tuple in `IS_CALLED` fail to match some tuple in `IS_ENROLLED_ON`:

```
      CONSTRAINT Student_must_be_enrolled_on_some_course
         IS_EMPTY ( IS_CALLED NOT MATCHING IS_ENROLLED_ON ) ;
```

  (This is *not* a foreign key constraint. Why not?)

The constraints for BR2 and BR4 together imply that at all times the set of `StudentId` values appearing in `IS_CALLED` must be equal to the set of `StudentId` values appearing in `IS_ENROLLED_ON`, so we could address those two constraints quite simply like this:

```
CONSTRAINT Being_enrolled_equivalent_to_having_a_name
   IS_CALLED { StudentId } = IS_ENROLLED_ON { StudentId } ;
```

Such a constraint is called an **equality dependency**.

As we have already seen, equality dependencies are a bit of a challenge for the DBMS and in most commercial DBMSs existing in 2009 it cannot even be expressed. However, in those same DBMSs the JD constraint required for `ENROLMENT` cannot be expressed either, so neither of the designs is fully implementable in the current technology! (Workarounds for SQL databases, using triggered procedures, are described in depth in reference [14].) If we *can* express both of those constraints, then, on the evidence so far, there doesn't seem to be a lot to choose between the two designs and our decision is more likely to be based on how well the DBMS supports multiple assignments; if we can't express the constraints, then the stated requirements cannot be met and we will have to compromise (and perhaps rely on application code to maintain integrity). But efficacy of constraint checking isn't the only criterion to guide our choice. What if, for example, student S1's name is incorrectly recorded as "Ann" and needs to be corrected to "Anne"? In the unnormalized design that correction will entail updates to several tuples in `ENROLMENT`, whereas in the 5NF design just one tuple in `IS_CALLED` is affected. On the other hand, a query to give the names of all the students enrolled on course C1 is simpler to express (and might run faster) in the unnormalized design. A frequent complaint about rigorous application of 5NF is that the decomposition causes too many joins to have to be used in queries. Advocates of 5NF respond by pointing out that the query to find the names of all the students is simpler to express (and might run faster) in the 5NF design!

It does seem that the 5NF design has a certain aesthetic appeal, giving a structure that is reduced to simpler terms and is thus in a sense more flexible. Also, an abiding motivation for the relational approach is that in principle the database designer need not anticipate the kinds of queries that will be presented to the DBMS. A 5NF design "levels the playing field" in keeping with that principle. Moreover, the discussion so far has been based on the assumption that the single relvar design, from which we inferred those business rules, is correct, an assumption we might well question.

In particular, we might question BR4: "Every student whose name is recorded is enrolled on at least one course". Does the university really require every student to be always enrolled on at least one course, even during the annual long vacation? What harm comes if that rule is relaxed? In that case the single relvar design becomes *incorrect*—and in any case we still need that complex and difficult constraint to express the JD. With the 5NF design the difficult equality dependency becomes replaced by a simple foreign key constraint to address BR2 and BR3.

Now, if 5NF is indeed a goal to be earnestly pursued, then some questions arise. How does the designer discover that a relvar under consideration is not in 5NF? How do we spot the rogue JDs—the nontrivial ones that do not arise simply as a consequence of keys? Well, there is an algorithm, given by Fagin (see the annotation for reference [13] in Appendix A) for determining whether a given JD is implied by the keys of the relvar to which it pertains, but as I mentioned earlier, some 30 years after the publication of reference [13] 5NF was discovered to be "too strong" for the purpose at hand. The details are in reference [8], which defines "Essential Tuple Normal Form" (ETNF). ETNF is both sufficient and necessary for the purpose at hand. Its definition is too complicated to be repeated here but happily it turns out to be easier to test for than 5NF, as we shall eventually see (the test depends on Boyce-Codd Normal Form, which is defined in Section 7.8).

Tests on JDs are all very well, but how can we determine the nontrivial JDs that hold in a proposed relvar? You have to examine the business rules and that's not always an easy task. However, a certain special kind of JD has been identified such that when it holds in a given relvar *r, r* is not in ETNF (and therefore not in 5NF). Not all ETNF-violating JDs are of this kind, but in practice most of them are. If we can eliminate such JDs we stand a reasonable chance of achieving ETNF by that action alone and any remaining rogue JDs should be quite easy to detect. A useful theory has been developed around this special kind of JD, making it possible to mechanize certain important aspects of relational database design.

This special kind of JD is one that is a direct consequence of another kind of constraint to which a relvar might be subject, called a *functional dependency*. There now follows a comparatively long dissertation on the concept—functional dependence—that surrounds functional dependencies, and, arising from that concept, keys. You need a good grasp of these topics before we can return to this special class of JD and discover the normal form that arises from it.

7.5 Functional Dependencies

Here again is the rogue JD that holds in ENROLMENT:

```
* { { StudentId, Name }, { StudentId, CourseId } }
```

Note the following points of significance:

1. It is a binary JD.
2. One of its two projections involves every attribute of a key of the relvar in which it holds.
3. The other projection, {StudentId, Name}, does not involve every attribute of that key (if it did, the JD would not be a rogue JD).
4. The relvar, IS_CALLED, arising from that other projection has a key that is a proper subset of its heading. (Strictly speaking a heading is a set of <attribute name, type name> pairs, but it is common practice to use the term to refer to a set of unaccompanied attribute names when no confusion can arise.)

Consider this last point. We can tell by inspection that it must hold, because otherwise IS_CALLED would represent a many-to-many mapping between student identifiers and names: the same student could have several names and several students could have the same name, in which case {StudentId, CourseId} would not be a key of IS_ENROLLED_ON. We can also tell that the proper subset in question must be {StudentId}. If instead it were {Name}, then again it would be possible for the same combination of StudentId and CourseId values to appear in more than one tuple of ENROLMENT, violating its given key constraint. So, in both designs we can say that no StudentId value appears in combination with more than one Name value in the relvar whose heading contains both of those attributes. In one of those relvars, ENROLMENT, the same combination can appear more than once; in the other, IS_CALLED, it cannot. The condition concerning student identifiers and names that holds in both of these relvars is called a **functional dependency**, denoted thus:

```
    { StudentId } → { Name }
```

The arrow in this notation is often pronounced "determines", so the whole expression can be pronounced "StudentId determines Name"—and so it does: given a student identifier we can determine the name that goes with it because there is only one such name. However, we shall soon see that "determines" doesn't always work so well and sometimes we have to fall back on just "arrow".

The reader with a mathematical bent will recognize that the set of <StudentId, Name> pairs constituting the body of the projection of ENROLMENT over those two attributes is a *function*, which justifies the chosen term, functional dependency. By convention and henceforth in this chapter we abbreviate it to FD. The FD { StudentId } → { Name } is said to *hold* in ENROLMENT, also in IS_CALLED. Equivalently, we say that relvars ENROLMENT and IS_CALLED each *satisfy* that FD.

Note that the left-hand side and right-hand side of an FD are both enclosed in braces, signifying that the enclosed elements constitute a set. The set on the left is called the *determinant*, that on the right the *dependant*. The term dependant is also used for each element of the right-hand side. For an example where more than one dependant appears on the right we need go no further than WIFE_OF_HENRY_VIII, in which the FD

```
    { Wife# } → { FirstName, LastName, Fate }
```

holds. `Wife#` determines each of `FirstName`, `LastName`, and `Fate`, so there are three FDs having the same determinant. Our notation allows them to be expressed as a single FD. A similar observation does not apply to the determinant. consider `EXAM_MARK`, for example, (Chapter 5, Figure 5.1). Its relvar definition is

```
VAR EXAM_MARK BASE RELATION { StudentId SID,
                              CourseId CID
                              Mark INTEGER }
                        KEY { StudentId, CourseId } ;
```

So the following FD holds in `EXAM_MARK`:

```
{ StudentId, CourseId } → { Mark }
```

Now we see why the pronunciation "arrow" is safer in general than "determines". The pronunciation "`StudentId`, `CourseId` determines `Mark`" doesn't work so well, but nor does making the verb plural to match its subject: "`StudentId`, `CourseId` determine `Mark`". The latter pronunciation might lead the listener to conclude, incorrectly, that `StudentId` determines `Mark` and `CourseId` determines `Mark`. For each pairing of a `StudentId` value with a `CourseId` value in `EXAM_MARK` there is exactly one mark, but the same student can obtain different marks for different courses and the same course can have different marks for different students. So neither of the FDs { `StudentId` } → { `Mark` } and { `CourseId` } → { `Mark` } holds in `EXAM_MARK`.

In the following formal definition for FD, note the parallels with the definition of superkey given in Chapter 6, Section 6.4 under the heading **Keys**. It uses both relational projection and tuple projection

Definition of FD

Let *A* and *B* be subsets of the heading of relvar *r*. Then the FD $A \to B$ holds in *r* if and only if, at all times, if tuples *t1* and *t2* both appear in the body of the projection $r\{A \cup B\}$, and the projection *t1*{*A*} is equal to the projection *t2*{*A*}, then *t1* = *t2* (they are the same tuple)

Given a set of FDs assumed to hold in relvar *r*, we can infer further FDs that must also hold in *r*. The *inference rules* used for this purpose are known as **Armstrong's Axioms** because they first appeared in a paper by Armstrong [2]. One way of expressing these rules is as follows. Let *A*, *B*, and *C* be arbitrary subsets of the heading of *r*. Then we have the following theorems (using the symbols "∪" and "–" for set union and set difference, respectively):

1. **Reflexivity:** If *B* is a subset of *A*, then $A \to B$
2. **Augmentation:** If $A \to B$, then $A \cup C \to B \cup C$
3. **Transitivity:** If $A \to B$ and $B \to C$, then $A \to C$

From these three we can derive:

4. **Self-determination:** $A \rightarrow A$
5. **Decomposition:** If $A \rightarrow B$ and C is a subset of B, then $A \rightarrow C$ and $A \rightarrow B\text{-}C$
6. **Union:** If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow B \cup C$
7. **Composition:** If $A \rightarrow B$ and $C \rightarrow D$, then $A \cup C \rightarrow B \cup D$

All of these except the first and fourth can be seen as special cases of

8. **Unification:** If $A \rightarrow B$ and $C \rightarrow D$, then $A \cup (C—B) \rightarrow B \cup D$

Given a relvar *r* and a set *S* of FDs assumed to hold in *r*, theorems 1 and 8 alone are sufficient to determine all the FDs in *r* that are implied by *S*. That set of FDs is called the **closure** of *S*, written as S^+.

Interesting Special Cases of FDs

In the following discussions, to save repetition, all FDs mentioned are assumed to hold in relvar *r* unless otherwise stated, and the symbol *C* refers to the set of all the FDs that hold in *r*. Now, in general *C* contains FDs that are redundant in the sense that they are implied under the given inference rules by other FDs in *C*. For example, *C* might contain `{a}` → `{b}`, in which case it must also contain `{a}` → `{a, b}`. We say that if *S1* and *S2* are sets of FDs and every FD in *S2* is implied by those in *S1*, then *S1* is a **cover** for *S2*. We are interested in finding proper subsets *S* of *C* where *S* is a cover for *C* and there is no proper subset of *S* that is also a cover for *C*. In that case, to enforce all the FDs in *C* it is sufficient for the DBMS just to enforce those in *S*—the others will all be enforced "automatically", as a logical consequence. Consideration of the special classes of FD that now follow helps us to find such sets. Most of the terms defined for these special classes are used conventionally in the literature, but I add some of my own.

Empty dependants

Because the determinant and dependant of an FD are both sets, we have to consider the possibility of either or both being the empty set, { }. Theorem 5, Reflexivity, tells us that for every subset *A* of the heading of *r, A* → { } holds in *r*. In particular { } → { } holds in *r*. If we eliminate from *C* all the FDs having empty dependants, then the resulting subset is clearly a cover for *C*.

Empty determinants

Much more interesting is the case, { } → *B*, where *B* is nonempty. This FD is satisfied only when each attribute *a* of *B* is such that *a* has the same value in each tuple of *r*—in other words, *a* is *constant* in *r*. If you need to be convinced of this fact, check it using the formal definition for FD given earlier. When *A* is the empty set, *t1*{*A*} and *t2*{*A*} are both the 0-tuple. It follows that every tuple of *r* has the same value (the 0-tuple) for the determinant of the FD and must therefore have the same value for the dependant.

Note that if some FD in *C* has an empty determinant and we eliminate from *C* all the FDs having empty determinants, then the resulting subset is *not* in general a cover for *C*.

Left-irreducible FDs

A → *B* is a **left-irreducible** FD if and only if there is no proper subset *A'* of *A* such that *A'* → *B* also holds. Conversely, if there is such a proper subset, then it is left-reducible. If we eliminate from *C* all those that are left-reducible, then from the resulting subset *S* we can recover *C* by application of Theorem 2, Left Augmentation. Thus, *S* is a cover for *C*.

Right-irreducible FDs

A → *B* is a **right-irreducible** FD if and only if *B* contains at most one attribute. If *B* contains more than one attribute, then the FD is right-reducible. If we eliminate from *C* all those that are right-reducible, then from the resulting subset *S* we can recover *C* by application of Theorem 6, Union. Thus, *S* is a cover for *C*.

Right-extendible FDs

$A \rightarrow B$ is a **right-extendible** FD if and only if there is some proper superset *B'* of *B* such that $A \rightarrow B'$ also holds. If we eliminate from *C* all those that are right-extendible, then from the resulting subset *S* we can recover *C* by application of Theorem 3, Decomposition. Thus, *S* is a cover for *C*.

Overlapping FDs

The FD $A \rightarrow B$ is **overlapping** if $A \cap B$ (where "$\cap$" denotes set intersection) is nonempty; otherwise it is non-overlapping. If we eliminate from *C* all those that are overlapping, then from the resulting subset *S* we can recover *C* by application of Theorem 6, Union. Thus, *S* is a cover for *C*.

Trivial FDs

The FD $A \rightarrow B$ is **trivial** if *B* is a subset of *A*; otherwise it is nontrivial. Note that if *B* is empty, then $A \rightarrow B$ is trivial but non-overlapping. If we eliminate from *C* all those that are trivial, then from the resulting subset *S* we can recover *C* by application of Theorem 5, Reflexivity. (Note that this elimination subsumes the elimination of FDs with empty dependants.) Thus, *S* is a cover for *C*.

Interesting Cases of Sets of FDs

We are interested in sets *S* of FDs, where no proper subset of *S* is a cover for the closure S^+ of *S*. In general there can be many such subsets of *S*, of which two kinds are particularly useful.

Irreducible covers

A set *S* of FDs is irreducible if and only if it satisfies the following three conditions:

1. Every FD is *S* is right-irreducible.
2. Every FD is *S* is left-irreducible.
3. No proper subset of *S* is a cover for the closure S^+ of *S*.

If *S* does satisfy those three conditions, then it is an **irreducible** cover for S^+. Note that condition 3 implies that *S* contains no overlapping FDs, no trivial ones, and none that are implied by others under Theorem 4, Transitivity. Some texts—[3], for example—use the term **nonredundant** for sets that satisfy condition 3 but do not necessarily satisfy conditions 1 and 2.

Minimal covers

Let *S1* be an irreducible cover for S^+. Let the set *S2* be derived from *S1* as follows. Wherever two or more FDs in *S1* have the same determinant *A*, replace those FDs by the single FD $A \rightarrow B$, where *B* is the union of the dependants of those FDs. Then *S2* is a **minimal cover** for S^+. (*Alert:* some texts use this term for irreducible covers.)

For example, the set { { `StudentId` } → { `Name` } } is both an irreducible cover and a minimal cover for all the FDs that hold in `ENROLMENT`. Note in particular that the FD { `StudentId`, `CourseId` } → { `Name` } follows, under Theorem 8, Unification, from { `StudentId` } → { `Name` } (given) and { `CourseId` } → { `CourseId` } (self-determination).

7.6 Keys

You might wish to read again the material under the heading **Keys** in Chapter 6, Section 6.4, which gives the following definitions:

Definitions for superkey and key

Let *K* be a subset of the heading of relvar *r*. Then *K* is **superkey** for *r* if and only if, at all times, if tuples *t1* and *t2* both appear in the body of *r*, and the projection *t1*{*K*} is equal to the projection *t2*{*K*}, then *t1* = *t2* (i.e., they are the same tuple).

K is a **key** for *r* if and only if (a) *K* is a superkey for *r* and (b) no proper subset of *K* is a superkey for *r*.

Appealing to FDs we can now give rather more concise definitions:

Let *K* be a subset of the heading *H* of relvar *r*. Then:

- *K* is a **superkey** of *r* if and only if FD $K \rightarrow H$ holds in *r*
- *K* is a **key** of *r* if and only if FD $K \rightarrow H$ is left-irreducible and holds in *r*

When a key is declared to the DBMS, only its uniqueness property is subsequently checked and maintained. Its irreducibility has to be the database designer's responsibility, for at any time *r* might be assigned a value that satisfies a putative key constraint implied by some proper subset of the declared key. For example, it is probably not a design error if at some point in time no two tuples in `IS_ENROLLED_ON` have the same `CourseId` value, even though the real world situation reflected by that state is perhaps a little surprising (no more than one student on any of the courses). On the other hand, it definitely would be a design error to include `Mark` along with `StudentId` and `CourseId` in the declared key for `EXAM_MARK`, allowing the same student to score several different marks in the same exam.

It must be emphasized that a key is a set of attributes, not an attribute per se. `EXAM_MARK` has just one key, consisting of two attributes, `StudentId` and `CourseId`. It is a common error to misunderstand the definition and refer to the attributes constituting the key as if each one were itself a key. In general a relvar can have several keys. For example, employees are identified in the company by their employee number but the database might record their national insurance number too, in some relvar that therefore has at least two keys, perhaps `{Employee#}` and `{NatIns#}`. Note the braces. Even though those two keys are both singleton sets, to omit the braces would (a) be sloppy and (b) perhaps foster the common misunderstanding I mentioned. On the other hand, the term "key attribute" is acceptable and useful.

Chapter 6 draws attention to two special cases of keys: the entire heading as a key, and the empty set as a key. Now you can see how the significance of the empty key arises from that of FDs having empty determinants.

7.7 The Role of FDs and Keys in Optimization

This section is a digression, inserted here to show you that database design is not the only area in which the theory of functional dependence is applicable. I take the opportunity to mention one other important application of the theory. It concerns optimization that a relational DBMS is expected to undertake in the interests of efficient evaluation of relational expressions, a process commonly called query optimization.

Just as the value assigned to a variable, by definition, varies from time to time, so also, as a consequence, does the relation denoted by a relational expression that references one or more relvars. Therefore our definition of "key" can be generalized to apply to relational expressions as well as to relvars. For example, given that { `StudentId`, `CourseId` } is a key of `EXAM_MARK`, we can conclude (for example) that { `StudentId` } is a key of (for example) `EXAM_MARK WHERE CourseId = CID('C1')`. Constraining `CourseId` values to be constant in the result of that expression means that the FD { } → { `CourseId` } holds in it, because application of Theorem 8, Unification, to { } → { `CourseId` } and { `StudentId`, `CourseId` } → { `Mark` } yields { `StudentId` } → { `Mark`, `CourseId` }. A subsequent projection over just `StudentId` and `Mark` (likely to be given because the constant `CourseId` value is known) can be performed efficiently just by dropping the `CourseId` attribute with no need to incur the overhead of detecting redundant duplicate tuples—there cannot be any—and eliminating them. If the DBMS can detect projections that preserve at least one key of the operand, then it can evaluate those projections more efficiently.

Now consider the expression `EXAM_MARK JOIN COURSE`. Because { `CourseId` } is a key of `COURSE` and `CourseId` is a common attribute for the join, there cannot be more than one matching tuple in `COURSE` for any given tuple of `EXAM_MARK`. When the DBMS, evaluating this join, discovers a matching `COURSE` tuple for a given `EXAM_MARK` tuple, it doesn't need to look for another one: thanks to its knowledge of keys, derived from its application of FD theory, it knows there cannot be one—and looking for something that doesn't exist usually takes rather longer than looking for and finding something that does!

That completes the discussion of functional dependence and we can now return to projection-join normalization, that special class of JDs that are implied by FDs. In so doing we will discover the normal form that arises from it, called **Boyce-Codd normal form** after Raymond F. Boyce and E.F. Codd who jointly first defined it, in [6].

7.8 Boyce-Codd Normal Form (BCNF)

Boyce-Codd normal form (BCNF) is important because (a) in practice, relvars that are in BCNF but not in 5NF are few and far between, and (b) we have a procedure for deriving, from a relvar that is not in BCNF, an equivalent set of relvars that *are* in BCNF, and that procedure can be automated (there is no known procedure for guaranteeing 5NF). BCNF is defined as follows.

Relvar *r* is in **Boyce-Codd normal form** (BCNF) if and only if and only if every functional dependency that is implied by the keys of *r*.

Equivalently:

Relvar *r* is in **Boyce-Codd normal form** (BCNF) if and only if the determinant of every nontrivial functional dependency that holds in *r* is a superkey of *r*.

That second definition effectively gives us a convenient test for an FD that isn't implied by keys. (The word nontrivial is needed here because all trivial FDs hold and don't all have superkeys as determinants. A trivial FD is "implied by keys" simply because it holds by virtue of itself alone.)

Recall that an FD $A \rightarrow B$ is trivial if and only if B is a subset of A. To test for BCNF it is sufficient to consider just the left-irreducible, non-overlapping FDs, whose determinants must then be not merely superkeys, but keys of *r*. For convenience, I use the term *rogue* FD for a left-irreducible, non-overlapping FD that holds in *r* but whose determinant is not a key of *r*.

I introduced the subject of functional dependence by showing the special kind of JD from which a corresponding FD can be derived. In practice the database designer will be looking for FDs that cause BCNF to be violated. When such an FD is discovered, the designer needs to be able to derive the corresponding JD to determine the decomposition needed to rectify the situation.

Let *A*, *B* and *C* be subsets of the heading of relvar *r* such that every attribute of *r* appears in at least one of those subsets and the FD $A \rightarrow B$ holds in *r*. Then, thanks to a theorem due to Heath [15], we can conclude that the JD $*\{A \cup B, A \cup C\}$ holds in *r*. Heath's theorem applies to relations, not relvars. It effectively states that as a consequence of the FD $A \rightarrow B$ holding in the current value of *r*, that value is equal to the join of its projections on $A \cup B$ and $A \cup C$. As that FD must hold in every value that *r* can take, we can conclude that the relvar itself is indeed subject to the JD $*\{A \cup B, A \cup C\}$.

Use of Heath's theorem is advocated in many texts for determining decompositions to yield relvars that are all in BCNF. In particular, if the following conditions on *A*, *B*, and *C* all hold, then *r* is not in BCNF:

- Every attribute of *r* is an element of exactly one of *A*, *B*, and *C*.
- *A* is not a key of *r*.
- *B* is nonempty.
- The FD $A \rightarrow B$ is nontrivial and left-irreducible.

In that case $A \rightarrow B$ is a rogue FD in *r* and a decomposition of *r* into relvars *s* with heading $A \cup B$ and *t* with heading $A \cup C$ can be considered. If that decomposition is taken, then $A \rightarrow B$ holds in *s* rather than *r* and, because *A* is a key of *s*, is no longer a rogue FD.

For example, consider `ENROLMENT` again. Then *A* = {`StudentId`}, *B*={`Name`}, and *C* = {`CourseId`}, giving us the decomposition we adopted into `IS_CALLED` with heading {`StudentId`, `Name`} and `IS_ENROLLED_ON` with heading {`StudentId`, `CourseId`}. In this case the two relvars resulting from the decomposition are both in BCNF, but it does not follow in general that *s* and *t* are in BCNF, for there might be other rogue FDs in *r*, propagated in *s* or *t* and remaining as rogues. In any case, when there is more than one rogue FD in *r*, decomposition into *s* and *t* is not always a well-advised step to take towards a design in which each relvar is in BCNF, as you will see in the next section, **Dependency preservation**.

It is easy to see that `ENROLMENT` is not in BCNF. The determinant, { `StudentId` }, of the only FD in its minimal cover of FDs is a proper subkey and therefore by definition not a superkey of `ENROLMENT`. Such FDs formed a basis for the definition, by Codd, of *second normal form* (2NF). Relvars in 2NF are not necessarily in BCNF but it was thought that decomposition to "just 2NF" might be a useful first step in a procedure to derive a complete BCNF design. Codd also defined *third normal form* (3NF), which covered some additional rogue FDs not covered by 2NF. 3NF was at one time thought to eliminate all cases of redundancy arising from FDs but Boyce showed that conclusion to be incorrect—hence the eventual arrival of BCNF. Some texts still advance 2NF and 3NF as useful steps on the way to BCNF but the rationale for that escapes the present author and this book shall have no more to say about those normal forms.

Dependency preservation

Consider relvar `SCDF` with attributes `S` (for student), `C` (for course), `D` (for department), and `F` (for faculty). Throughout the following discussion relvar names reflect the relevant attribute names in similar fashion, allowing a relvar's heading to be inferred from its name for convenience. Assume that the set {{`C`} → {`D`}, {`D`} → {`F`}} is a minimal cover for the FDs in `SCDF`. Then {`S`,`C`} is a key of `SCDF` (*exercise:* prove that) and `SCDF` is clearly not in BCNF. If we apply Heath's theorem to the FD {`C`} → {`D`}, we will decompose `SCDF` into relvars `CD` and `SCF` with keys {`C`} and {`S`,`C`}, respectively. `CD` is in BCNF but `SCF` is not. Applying Heath's theorem to the rogue FD in `SCF`, we obtain relvars `CF` and `SC`, both in BCNF. We now have three relvars, all in BCNF, but no relvar whose heading includes both `D` and `F`. As a consequence, the constraint represented by the FD {`D`} → {`F`} cannot be expressed as a constraint on a single relvar in this BCNF design—the FD has been "lost" in the decomposition. On the other hand, if instead we start by applying Heath's theorem to the FD {`D`} → {`F`}, then the resulting relvars are `DF` and `SCD` with keys {`D`} and {`S`,`C`}, respectively. The FDs of the given cover are both "preserved" in this decomposition, {`D`} → {`F`} in `DF` and {`C`} → {`D`} in `SCD`. `DF` is in BCNF but `SCD` is not. Applying Heath's theorem to the rogue FD in `SCD`, we obtain relvars `CD` and `SC`, both in BCNF. This time, both of the original FDs are preserved in the decomposition.

A conclusion from the foregoing observations is that if repeated application of Heath's theorem is the procedure adopted for achieving BCNF designs, then sometimes one has to take care of the order in which binary decompositions are taken if one wishes to avoid needlessly complex constraints. The problem is neatly avoided, *where possible,* in the following procedure (a simplification of one given by Date in [9], Chapter 12), which guarantees to preserve all FDs but in some comparatively rare cases yields relvars that are still not in BCNF. Let *r* be a relvar and let *S* be a minimal cover for the FDs that hold in *r*. Then:

1. Initialize *D* to the empty set.
2. For each FD $X \rightarrow Y$ in *S,* form the relvar *XY* with heading $X \cup Y$ and replace *D* by the union of *D* and {*XY*}.
3. If no relvar in *D* includes in its heading some key of *r*, then let *K* be a relvar whose heading consists of the attributes of some key of *r,* arbitrarily chosen, and replace *D* by the union of *D* and {*K*}.

Under Date's procedure each FD in *S* is guaranteed to be preserved in some relvar in *D*. However, there might be some relvars in *D* that are still not in BCNF—we will come to the circumstances in which these can arise later. To derive a complete BCNF design from *D* we must add the following step:

4. If *D* contains any relvars that do not satisfy BCNF, then:
   a) Let *T* be any one of those relvars, let *H* be the heading of *T*, and let $X \rightarrow Y$ be a rogue FD in *T*.
   b) Replace *T* in *D* by two relvars, *T'* resulting from the projection of *T* over $X \cup Y$ and *T"* resulting from the projection of *T* over *H*-*Y* (an application of Heath's Theorem). Superkeys of *T* that do not contain any attribute of *Y* are superkeys of *T'* and *X* is a superkey of *T"*.
   c) Declare constraints as needed to compensate for any lost FDs.
   d) Repeat Step 4.

All of the relvars produced by this algorithm are guaranteed to be in BCNF, but unfortunately they are not guaranteed to be in 5NF or even ETNF—we will look at some examples later in this chapter (Section 7.9, **JDs Not Arising from FDs**).

One could also work from an irreducible cover instead of a minimal one. In an irreducible cover every FD has exactly one attribute on the right-hand side (the dependant). Thus, if we work from an irreducible cover, every resulting relvar that is in 5NF will also be in 6NF. As the discipline of projection-join normalization sometimes leads the designer to question the given assumptions (such as requiring every student to be enrolled on at least one course), perhaps that approach should be considered: start with an irreducible cover and add Step 5, as follows, noting, however, that this one does not obviously lend itself to mechanization.

5. If certain relvars in *D* have some key in common, consider joining some of them together with a view to eliminating constraints needed to express equality dependencies.

For example, application of Date's procedure, as stated, to `WIFE_OF_HENRY_VIII` would result in no change: the relvar is already in BCNF (and, as it happens, 5NF). Applying it to an irreducible cover instead of a minimal cover would yield the three relvars shown in Design B, discussed in the section following Figure 7.3, headed **Assessment of 6NF Decomposition.** Application of the suggested Step 6 might indicate combining `W_FN` and `W_LN` as `W_FN_LN` but keeping `W_F` separate.

Now we must look at the circumstances under which Step 4 is needed, and why an FD is inevitably lost in that step.

FD Loss Sometimes Inevitable

Consider the relvar `WXYZ` with attributes `W`, `X`, `Y`, and `Z`. Assume that {{`W`,`X`} → {`Y`,`Z`}, {`Y`} → {`X`}} is a minimal cover for `WXYZ`. Then {`W`,`X`} and {`W`,`Y`} are both keys of `WXYZ` (*exercise:* prove that claim), and `WXYZ` is not in BCNF because the determinant {`Y`} is not a superkey. Date's procedure (Steps 1 to 3) yields the BCNF relvar `XY` with key {`Y`} but retains the non-BCNF relvar `WXYZ`. In Step 4 we decompose `WXYZ` into `WXZ` and `XY`, now both in BCNF, but then we lose the FD {`W`,`X`} → {`Y`}. We will need to declare a constraint to the effect that {`W`,`X`} is a key of `WXZ JOIN XY`.

For a realistic example exhibiting this phenomenon, consider payments made against bank accounts by use of debit cards. Payments are identified within a particular account by transaction numbers, so payments by debit card might be recorded in a relvar with attributes `Account#`, `Transaction#`, `Card#`, `Payee`, and `Amount`. {`Account#, Transaction#`} is a key for this relvar but so is {`Transaction#, Card#`} because we have {`Card#`} → {`Account#`}. (There might be several distinct cards for the same account: for example, if it is a joint account.)

In general, if *A*, *B*, *C*, and *D* are subsets of the heading of relvar *r* and FDs *A*4*B*→*C*4*D* and *C*→*B* hold in *r*, then *A*4*B* and *A*4*C* are both keys of *r*. If each attribute of *r* appears in exactly one of *A*, *B*, *C*, and *D*, *A* and *B* are each nonempty, and either *C* or *D* is nonempty, then

a) *C*, a proper subkey, is the determinant of a nontrivial FD and *r* is therefore not in BCNF.
b) Steps 1 to 3 will not result in a BCNF decomposition of *r*.
c) The nontrivial FD *A*∪*B*→*C*∪*D* is lost in the decomposition resulting from Step 5.

One might even question the claim that Steps 1 to 4 guarantee to preserve FDs. Consider the relvar SCF with attributes S, C, and F, standing for student, course, and faculty as before. This time, however we have the FD {S} → {F} in addition to {C} → {F}. Our minimal cover is therefore {{S} → {F}, {C} → {F}}, from which we can conclude that {S,C} → {F} holds in SCF and {S,C} is a key of SCF (*exercise:* prove this). Application of Step 2 yields relvars SF and CF and Step 4 yields SC. Have we lost the FD {S,C} → {F}, which, although not present in the given minimal cover, is a consequence of that cover? In one sense, no, because it holds in JOIN{SF, CF, SC}, but the fact remains that the decomposition allows a student to be enrolled on a course offered by a faculty other than the student's faculty unless some constraint is declared to make that impossible. With relvar SCF that was not possible. If we consider UNION{JOIN{SC,SF}, JOIN{SC,CF}} we can discover a sense in which that FD has indeed been lost, for that can yield a relation in which it does not hold. As with the 6NF decomposition of WIFE_OF_HENRY_VIII we have a situation where a constraint implied by the very structure of a relvar has to be stated explicitly when a decomposition is taken.

Another point about the BCNF decomposition of `SCF` is that if the single relvar design is a correct implementation of the requirements, then we still have redundancy after the decomposition! For on that assumption of correctness we can infer the following business rules:

BR1: A student can be enrolled on several courses.
BR2: Several students can be enrolled on a course.
BR3: At least one student is enrolled on every course.
BR4: Every student is enrolled on at least one course.
BR5: Every student belongs to exactly one faculty.
BR6: Every course is offered by exactly one faculty.
BR7: Students can only be enrolled on courses offered by their own faculty.

Normalizing to BCNF will yield the design shown in Example 7.3.

Example 7.3: Decomposing SCF into SF, CF, and SC

```
VAR SF BASE RELATION { S SID,
                       F CHAR }
                 KEY { S } ;

VAR CF BASE RELATION { C CID,
                       F CHAR }
                 KEY { C } ;

VAR SC BASE RELATION { S SID,
                       C CID }
                 KEY { S, C } ;

CONSTRAINT Only_known_students_can_be_enrolled
    IS_EMPTY ( SC NOT MATCHING SF ) ;

CONSTRAINT Only_known_courses_can_be_enrolled_on
    IS_EMPTY ( SC NOT MATCHING CF ) ;

CONSTRAINT Every_Student_Enrolled
    IS_EMPTY ( SF NOT MATCHING SC ) ;

CONSTRAINT At_least_one_student_on_each_course
    IS_EMPTY ( CF NOT MATCHING SC ) ;
```

```
CONSTRAINT Same_Faculty_for_Student_and_Course
    IS_EMPTY ( JOIN { SC, SF RENAME ( F AS Sfac ),
                          CF RENAME ( F AS Cfac ) }
               WHERE NOT ( Cfac = Sfac ) ) ;
```

But we can obtain a student's faculty from `SC JOIN CF`, so `SF` is completely redundant and the design can be simplified as shown in Example 7.4.

Example 7.4: Decomposing SCF into just CF and SC

```
VAR CF BASE RELATION { C CID,
                       F CHAR }
                 KEY { C } ;

VAR SC BASE RELATION { S SID,
                       C CID }
                 KEY { S, C } ;

CONSTRAINT Only_known_courses_can_be_enrolled_on
    IS_EMPTY ( SC NOT MATCHING CF ) ;

CONSTRAINT At_least_one_student_on_each_course
    IS_EMPTY ( CF NOT MATCHING SC ) ;

CONSTRAINT FD_student_id_determines_faculty
    COUNT ( ( CF JOIN SC ) { S } ) =
    COUNT ( ( CF JOIN SC ) { S, F } ) ;
```

Alternatively, we can obtain the faculty offering a course from `SF JOIN SC`, allowing us to eliminate `CF`, as shown in Example 7.5.

Example 7.5: Decomposing SCF into just SF and SC

```
VAR SF BASE RELATION { S SID,
                       F CHAR }
                 KEY { S } ;

VAR SC BASE RELATION { S SID,
                       C CID }
                 KEY { S, C } ;
```

```
CONSTRAINT Only_known_students_can_be_enrolled
     IS_EMPTY ( SC NOT MATCHING SF ) ;

CONSTRAINT Every_Student_Enrolled
     IS_EMPTY ( SF NOT MATCHING SC ) ;

CONSTRAINT FD_course_id_determines_faculty
     COUNT ( ( SF JOIN SC ) { C } ) =
     COUNT ( ( CF JOIN SC ) { C, F } ) ;
```

So, if those business rules are really correct, then we have a choice of BCNF designs, one of which (Example 7.3) involves redundancy (though not redundancy resulting from FDs) and the other two of which appear to be equal in merit (and significantly less complicated than Example 7.3). But rules **BR3** and **BR4** are surely open to question. If **BR3** is correct but **BR4** is not—it is possible, after all, for a student to exist who isn't enrolled on any course—then we can choose between Example 7.3, without the constraint `Every_Student_Enrolled`, and Example 7.4. If **BR4** is correct but **BR3** is not—it is possible, after all, for a course to exist on which no student is enrolled—then we can choose between Example 7.3, dropping `At_least_one_student_on_each_course`, and Example 7.5. But if **BR3** and **BR4** are both incorrect, then we have to go for the simplification of Example 7.3 shown in Example 7.6.

Example 7.6: Simplification of Example 7.3, relaxing some constraints

```
VAR declarations as in Example 7.3
CONSTRAINT Only_known_students_can_be_enrolled
     IS_EMPTY ( SC NOT MATCHING SF ) ;
CONSTRAINT Only_known_courses_can_be_enrolled_on
     IS_EMPTY ( SC NOT MATCHING CF ) ;
CONSTRAINT Same_Faculty_for_Student_and_Course
     IS_EMPTY ( JOIN { SC, SF RENAME ( F AS Sfac ),
                           CF RENAME ( F AS Cfac ) }
                  WHERE NOT ( Cfac = Sfac ) ) ;
```

Of course, Example 7.6 still involves some of the redundancy exhibited in Example 7.3. If a student *is* enrolled on some courses, then the faculty offering those courses must be that student's faculty; and if some students *are* enrolled on a course, then the faculty offering that course must be the faculty of those students. If we wish to avoid such redundancy, then the constraints become simpler as Example 7.7 shows.

Example 7.7: Complication of Example 7.6, avoiding redundancy

```
VAR  declarations as in Example 7.3
CONSTRAINT SF_only_for_students_not_enrolled
     IS_EMPTY ( SF MATCHING SC ) ;
CONSTRAINT SC_only_for_courses_nobody_enrolled_on
     IS_EMPTY ( CF MATCHING SC ) ;
```

But with that design we cannot even express the constraint requiring students to be enrolled only on courses offered by their own faculty. Moreover, updating becomes complicated. When a student gets enrolled on their very first course we have to use multiple assignment to delete a tuple from SF at the same time as adding one to SC. Similarly, when a course gets its very first enrolment we have to use multiple assignment to delete a tuple from CF at the same time as adding one to SC. And we have to reverse those processes when a student disengages from their only remaining course, and when a course loses its last student.

It seems that redundancy is not something to be avoided at all costs. Sometimes "all costs" do outweigh the advantages, as would surely be our judgment in the case of Example 7.7. Also, consider how the chosen design might affect the confidence that users have in the correctness of the database. With Example 7.6 one can be pretty sure of no student/course mismatches on faculty—they could arise only when a student's faculty or a course's faculty is incorrectly recorded in SF or CF. However, redundancy arising from join dependencies does appear to be worth avoiding, in general, *so long as the DBMS lets you declare all the constraints that are needed*. If it doesn't, then perhaps somebody should be complaining—after all, BCNF and its ramifications have been well understood for over 30 years at the time of writing.

One little topic remains to be covered before we can leave this chapter: join dependencies that do *not* arise from FDs. We haven't seen any of those yet. There are some, and these are the JDs that can cause a BCNF relvar not to satisfy ETNF (and 5NF).

7.9 JDs Not Arising from FDs

JDs that do not arise from FDs arise most commonly in "all key" relvars—relvars whose only key consists of the entire heading, such as `IS_ENROLLED_ON`, for example. A minimal FD cover for such a relvar is the empty set. For if *A*, *B*, and *C* are subsets of the heading of relvar *r* and { $A \rightarrow B$ } (a set consisting of just one FD) is a minimal cover for *r*, then $A \cup C$ must be a key of *r* and *B* is nonempty by definition of minimal cover. If *B* is nonempty, then $A \cup C$ is a proper subset of the heading and *r* is not "all key".

Clearly, an all-key relvar must be in BCNF. Now consider Example 7.8

Example 7.8: SCT: a relvar in BCNF but not in 5NF

```
VAR SCT BASE RELATION { S SID,
                        C CID,
                        T CHAR }
                     KEY { ALL BUT } ;
```

Predicate: Student *S* studies topic *T* on course *C*.

The minimal FD cover for `SCT` is empty—no right-irreducible FD holds in `SCT` whose determinant consists of two of those three attributes. Yet the JD *{{`S`,`C`}, {`C`,`T`}} holds in `SCT` on the assumption (let us assume it is safe) that a student enrolled on a course is studying every topic covered by that course. In other words, if student *s1* is shown by some tuple in `SCT` to be studying topic *t1* on course *c1,* and student *s2* is shown by some tuple in `SCT` to be studying topic *t2* on course *c1,* then a tuple showing student *s1* to be studying *t2* on course *c1* must also appear in `SCT`—otherwise the constraint implied by the given JD would not be satisfied. A decomposition into relvars `SC` and `CT` (with attributes as indicated by the relvar names, as before) is clearly indicated. `SCT` is in BCNF but not 5NF. `SC` and `CT` are both in 5NF.

Observations similar to those made in connection with decompositions indicated by FDs apply here too. If the decomposition into `SC` and `CT` really is required to be equivalent to `SCT`, then every course that covers at least one topic must have at least one student on it, and every course on which at least one student is enrolled must cover at least one topic. A constraint with condition equivalent to `SC{C} = CT{C}` needs to be declared. But the putative business rule giving rise to a requirement for such a constraint is questionable. Can't a course exist before any students are enrolled on it? Does every course have to be subdivided into topics? The decomposition allows enrolments and coverage of topics to be recorded independently of each other—which seems intuitively sensible in any case.

Now, when BCNF was first introduced it was known that relvars like `SCT` existed that are in BCNF and yet exhibit redundancy that can be avoided by taking projections. It was some time later that these cases were carefully studied, the concept of JD was formulated as a generalization of FD, and even stronger normal forms were defined to fill the remaining gaps left by BCNF. As 2NF and 3NF have already been mentioned as concepts previously essayed but later superseded (by BCNF), you must be wondering about the apparent numerical gap between 3NF and 5NF. Well, `SCT` is in fact in violation of 4NF as well as 5NF. Observe that the JD *{{`S`,`C`}, {`C`,`S`}} that causes `SCT` to violate 5NF is a *binary* JD in particular. It is because that JD is binary that `SCT` violates 4NF. 4NF was originally defined, not in terms of JDs in general, but in terms of a kind of dependency called *multi-valued* dependency (MVD), of which a functional dependency is a special case. However, it turns out that every MVD is equivalent to a certain binary JD, so we no longer really need to study MVDs—the interested reader can easily find out about them in the literature. Suffice it here just to mention that 4NF is weaker than ETNF and therefore does not guarantee to avoid redundancy.

It follows from the definition of 4NF that any relvar that is in 4NF but not 5NF must be subject to a nonredundant JD of degree greater than two (where a JD is redundant if and only if at least one of its projections is redundant). Moreover, that JD must be such that it's not merely a consequence of two or more binary JDs, unlike the nontrivial ternary JD we noted in `WIFE_OF_HENRY_VIII`. A relvar subject to such a JD cannot be decomposed into 5NF relvars by a succession of two-way decompositions—again, unlike `WIFE_OF_HENRY_VIII`, which we decomposed in two stages. The study of such relvars eventually led to 4NF being subsumed by 5NF (and much later by ETNF). We must now complete the story of projection-join normalization by showing that such JDs (let's call them *irreducible non-binary* JDs), and therefore such relvars, do indeed exist.

Irreducible non-binary JDs

Here again is the definition I gave earlier for fifth normal form:

> Relvar *r* is in **fifth normal form** (5NF) if and only if every join dependency that holds in *r* is implied by the keys of *r*.

We can retrospectively define 4NF just by inserting the word "binary" immediately before "join" in the definition of 5NF. The JD *{{`S`,`C`}, {`C`,`T`}} holds in `SCT` and it is intuitively clear that, given that the only key of `SCT` is the entire heading, we cannot deduce from that fact alone that this JD holds (as is confirmed by application of the algorithm given under reference [13] in Appendix A).

To discover a case of "4NF but not 5NF", we need to find a relvar that is subject to an irreducible non-binary JD. Well, consider Example 7.9, a relvar used by the university library to record which books are recommended by which lecturers for which topics on which courses.

Example 7.9: LBTC: a relvar to illustrate irreducible JDs

```
VAR LBTCS BASE RELATION { L LID,
                          B ISBN,
                          T CHAR,
                          C CID }
                      KEY { ALL BUT } ;
```

Predicate: Lecturer *L* recommends book *B* for topic *T* on course *C*.

The minimal FD cover for `LBTC` is empty—no right-irreducible FD holds in `LBTC` whose determinant consists of three of those four attributes.

Suppose a business rule is applicable, stating that if

- a lecturer recommends a certain book at all, and
- the book covers a certain topic, and
- the lecturer in question teaches that topic on a certain course,

then it follows that that lecturer recommends that book for that topic on that course. In that case the ternary JD *{{L,B},{B,T},{L,T,C}} holds in `LBTC`, but no nontrivial binary JD holds in it: `LBTC` is in 4NF but not in 5NF, and a three-way decomposition into `LB`, `BT` and `LTC` is indicated.

Note the difference in kind between the ternary JD *{{L,B},{B,T},{L,T,C}} and the one we found holding in WIFE_OF_HENRY_VIII:

```
* { {Wife#, FirstName}, {Wife#, LastName}, {Wife#, Fate} }
```

First, note that application of Fagin's algorithm shows that this JD is "implied by the keys" of WIFE_OF_HENRY_VIII and so does not give rise to a violation of 5NF. Secondly, we can take the union of any two of those projections to form a nontrivial binary JD that also holds in WIFE_OF_HENRY_VIII, for example:

```
* { { Wife#, FirstName }, { Wife#, LastName, Fate } }
```

Now, the JD *{{L,B},{B,T},{L,T,C}} in LBTC arose from a certain assumed business rule. Variations on that assumption lead to different JDs, necessitating different decompositions. For example, the business rule might instead state that if

- a lecturer recommends a certain book at all, and
- the book covers a certain topic, and
- the topic is included in a certain course,

then it follows that that lecturer recommends that book for that topic on that course. In that case the ternary JD *{{L,B},{B,T},{T,C}} holds in LBTC and a decomposition into LB, BT, and TC is indicated.

Or perhaps the business rule states, more strongly, that if

- a lecturer recommends a certain book at all, and
- the book covers a certain topic, and
- the topic is included in a certain course, *and*
- *the lecturer teaches on that course,*

then it follows that that lecturer recommends that book for that topic on that course. In that case the *quaternary* JD *{{L,B},{B,T},{T,C},{L,C}} holds in LBTC (even though no ternary JD holds) and a decomposition into LB, BT, TC, and LC is indicated.

Or perhaps the *quinary* JD *{{L,B},{B,T},{L,T},{T,C},{L,C}} holds, requiring the lecturer not only to teach the relevant course but also to teach the relevant topic (though not necessarily on that course).

Finally, what if the rule, albeit quite unreasonable, were that if a book covers a certain topic and that topic is included in a certain course, then it can be assumed that *every* lecturer recommends that book for that topic on that course. Then the JD is *{{L},{B,T,C}} and the join is in fact a Cartesian product—we might call the JD a "times dependency". We are led to the observation, interesting from a theoretical point of view, that even a binary relvar can be subject to a nontrivial join dependency. Suppose, for example, that a certain insurance company offers a product that insures its policy holders for driving in every country in the European Union. Then we don't need a relvar that pairs every policy holder with every EU member state!

Why ETNF is preferred

Considering the large number of JDs that might need to be examined in connection with relvars of even quite small degree, we can be grateful to the science offered by functional dependency analysis for giving us ways of getting to grips with the vast majority of them. It seems that when a BCNF design is achieved, no algorithmic procedure has yet been devised to yield a further decomposition that guarantees 5NF. But what about ETNF? It turns out that, although, as I stated earlier, the definition of ETNF is too complicated for this book, there is in fact a simple test for it, an equivalent definition that reference [8] calls a "syntactic characterization". At last we have covered all the material that is needed to understand this definition, so here it is:

> Relvar *r* is in **essential tuple normal form** (ETNF) if and only if *r* is in BCNF and some component of every nontrivial join dependency that holds in *r* is a superkey of *r*.

As you can see, it is somewhat similar to the second definition I gave for BCNF:

> Relvar *r* is in **Boyce-Codd normal form** (BCNF) if and only if the determinant of every nontrivial functional dependency that holds in *r* is a superkey of *r*.

Recall that under Heath's theorem, if *A*, *B* and *C* are subsets of the heading of relvar *r* such that every attribute of *r* appears in at least one of those subsets, then the FD $A \rightarrow B$ holding in *r* implies that the JD $*\{A \cup B, A \cup C\}$ holds in *r*. If *A* is a superkey of *r*, then that FD satisfies BCNF and that JD satisfies ETNF.

By the way, it is also shown in [8] that if *A4B* is a superkey of *r* and JD $*\{A \cup B, A \cup C\}$ holds, then FD $A \rightarrow B$ holds. In other words, if we extend Heath's theorem to include a statement of the superkeys of the projections mentioned in the JD, then the converse also holds. Most textbooks tell you, correctly, that the converse of Heath's theorem as stated does not hold.

ETNF doesn't guarantee 5NF, but we don't *need* 5NF for avoiding the kind of redundancy in question here. ETNF is sufficient (and necessary!). To obtain an ETNF decomposition of a given relvar we apply the procedure for obtaining BCNF relvars and then look for nontrivial JDs that are not implied by FDs. If we find one in which no component is a superkey, then we have an ETNF violation on our hands and decompose accordingly, repeating the process until all relvars are in ETNF.

Isn’t there a 1NF to precede 2NF?

I have described, or at least mentioned, projection-join normal forms numbered from 2 to 6 that, among others, have been defined over the years, but surely the numbering doesn’t start at two? What happened to *first* normal form, 1NF? That NF has indeed been defined but it is not a projection-join normal form and so does not belong in this chapter. It is discussed in the first section of Chapter 8.

EXERCISES

1. (Repeated from the body of the chapter). The predicate for `WIFE_OF_HENRY_VIII` is “The first name of the *Wife#*-th wife of Henry VIII is *FirstName* and her last name is *LastName* and *Fate* is what happened to her.” Write an appropriate predicate for the following expression:

```
WIFE_OF_HENRY_VIII { Wife#, FirstName }
JOIN
WIFE_OF_HENRY_VIII { LastName, Fate }
```

2. Consider the following declarations:

```
VAR C1_EXAM_MARK BASE
     INIT ( EXAM_MARK WHERE CourseId = CID('C1') )
      KEY { StudentId } ;
CONSTRAINT C1_only
     AND ( C1_EXAM_MARK, CourseId = CID('C1') ) ;
```

   a) Explain why `C1_EXAM_MARK` is not in BCNF.
   b) Assume that similar relvars are defined for every course, except that this time there are no `CourseId` attributes. Describe how a query could be expressed to give the course identifier and mark for every exam taken by student S1.
3. In Section 7.5 of the chapter, under the heading **Functional Dependencies**, eight theorems are given concerning FDs. Taking the first three as axioms, prove theorems 4 to 8.
4. (Repeated from the body of the chapter). Consider relvar `SCDF` with attributes `S` (for student), `C` (for course), `D` (for department), and `F` (for faculty). Assuming that the set {{`C`} → {`D`}, {`D`} → {`F`}} is a minimal cover for the FDs in `SCDF`, prove that {`S`,`C`} is a key of `SCDF`.
5. (Repeated from the body of the chapter). Assume that {{`W`,`X`} → {`Y`,`Z`}, {`Y`} → {`X`}} is a minimal FD cover for the FDs in relvar `WXYZ`. Prove that {`W`,`X`} and {`W`,`Y`} are both keys of `WXYZ`.
6. The heading of relvar `R1` consists of attributes named `a`, `b`, `c`, `d`, `e`, `f`, `g`, and `h`. The following set of FDs is a cover for those that hold in `R1`:

FD1:	{a,b} → {c}
FD2:	{a,b} → {d}
FD3:	{b} → {e}
FD4:	{c} → {f}
FD5:	{g} → {h}
FD6:	{d} → {b, e}

   a) Describe the single change required to derive an irreducible cover from the given set.
   b) Describe the single change required to derive a minimal cover from your answer to a.
   c) Explain why `R1` is not in Boyce-Codd normal form (BCNF).
   d) Decompose `R1` into an equivalent set of BCNF relvars. Name your relvars `R2`, `R3`, and so on and for each one list its attribute names and state its key(s). For example: `R3{c,d,e} KEY{d} KEY{c,e}` if you think this relvar with those two keys is part of the solution.

7. Write a complete **Tutorial D** definition for the database concerning payments made against bank accounts, described in Section 7.8, **Boyce-Codd Normal Form (BCNF)**. Include any other kinds of transaction you can think of in addition to those mentioned in Section 7.8—for example, you might wish to include payments into the accounts as well as payments out. Include also relvars to record details of accounts, customers, and debit cards. Write down business rules, in the style adopted in the chapter, stating the assumptions on which your design is based.
8. Based on your experiences with Exercise 7, suggest enhancements to **Tutorial D** to make it easier to express any constraints you declared that struck you as being of a common enough kind to warrant an additional shorthand.
9. (For students familiar with SQL). Consider the following SQL definitions:

```
CREATE TABLE SF ( StudentId CHAR(4),
                  Faculty VARCHAR(50),
       PRIMARY KEY ( StudentId )
       UNIQUE ( StudentId, Faculty ) ;
CREATE TABLE CF ( CourseId CHAR(4),
                  Faculty VARCHAR(50),
       PRIMARY KEY ( CourseId )
       UNIQUE ( CourseId, Faculty );
CREATE TABLE SCF ( StudentId CHAR(4),
                   CourseId CHAR(4),
                   Faculty VARCHAR(50),
       PRIMARY KEY ( StudentId, CourseId ),
       FOREIGN KEY ( StudentId, Faculty )
                    REFERENCES SF ( StudentId, Faculty ),
       FOREIGN KEY ( CourseId, Faculty )
                     REFERENCES CF ( CourseId, Faculty ) ;
```

a) What problem was the designer solving here?
b) What possible problem remains in this solution?
c) Describe and comment on the particular features of SQL that make this solution possible.

8 Database Design II: Other Issues

8.1 Group-Ungroup and Wrap-Unwrap Normalization

Preliminary remarks

I should make it clear right away that to most people the term "normalization", in the context of databases, means projection-join normalization specifically. But the term does seem to be equally appropriate in connection with other kinds of design choice, hence my decision to use it in the first two sections of this chapter.

I promised a discussion of 1NF in this section. It is here, but later.

Some examples in this chapter refer to the relvars `COURSE` and `EXAM_MARK` introduced at the beginning of Chapter 5. Their current values are repeated here in Figure 8.1.

COURSE

CourseId	Title
C1	Database
C2	HCI
C3	Op systems
C4	Programming

EXAM_MARK

StudentId	CourseId	Mark
S1	C1	85
S1	C2	49
S1	C3	85
S2	C1	49
S3	C3	66
S4	C1	93

Figure 8.1: Current values of relvars COURSE and EXAM_MARK

And here again are the relvar definitions for `COURSE` and `EXAM_MARK`:

```
VAR COURSE BASE RELATION { CourseId CID, Title CHAR }
                     KEY { CourseId };
VAR EXAM_MARK BASE RELATION { StudentId SID, CourseId CID,
                              Mark INTEGER }
                     KEY { StudentId, CourseId };
```

Group-ungroup normalization

In Chapter 5, Example 5.12 illustrates the **Tutorial D** operator `GROUP`. The expression `EXAM_MARK GROUP { StudentId, Mark } AS ExamResult`, operating on the current value of `EXAM_MARK`, yields the relation shown in Figure 5.6, repeated here as Figure 8.2 for convenience:

CourseId	ExamResult	
C1	StudentId	Mark
	S1	85
	S2	49
	S4	93
C2	StudentId	Mark
	S1	49
C3	StudentId	Mark
	S3	66

Figure 8.2: A relation derived from EXAM_MARK by grouping

Recall also that we can reverse the process by use of `UNGROUP`. Thus, a relvar defined as in Example 8.1 might be considered as a valid alternative to `EXAM_MARK`, but note the constraints needed to make the "grouped" design genuinely equivalent.

Example 8.1: Alternative design for exam marks, using a relation-valued attribute

```
VAR C_ER BASE
    INIT ( EXAM_MARK GROUP {StudentId, Mark} AS ExamResult )
    KEY { CourseId } ;

CONSTRAINT At_least_one_mark_per_course
    IS_EMPTY ( C_ER WHERE IS_EMPTY (ExamResult ) ) ;

CONSTRAINT At_most_one_mark_per_student_per_course
    IS_EMPTY ( C_ER WHERE COUNT (ExamResult{StudentId}) <
                          COUNT (ExamResult) );

CONSTRAINT CourseId_foreign_key
    IS_EMPTY ( C_ER NOT MATCHING COURSE ) ;

CONSTRAINT StudentId_foreign_key
    IS_EMPTY ( ( C_ER UNGROUP ExamResult )
               NOT MATCHING IS_CALLED ) ;
```

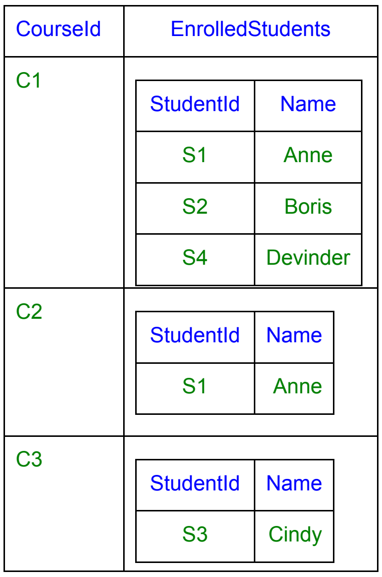
(The definition of relvar C_ER above is able to omit the type specification because in **Tutorial D** the declared type of a variable can be omitted when an `INIT` specification is included. The declared type of the variable is then that of the specified expression.)

The first of those two constraints reflects the fact that EXAM_MARK by itself cannot accommodate a course for which nobody sat the exam. It would probably make better sense to disregard that putative requirement and drop the constraint. The second constraint is a key constraint on the attribute values for `ExamResult`, a logical consequence of `KEY { StudentId, CourseId }` specified for `EXAM_MARK`. The `KEY` shorthand could be considered for relation-valued attributes of relvars but it is not included in **Tutorial D**, one reason being that such a design is in general contraindicated and should be discouraged. Here are some points against it:

1. The particular grouping chosen is arbitrary. Why not group on `{ CourseId, Mark }` instead?
2. The constraints are more complicated, even if we drop the one requiring `ExamResult` values to be nonempty.
3. Updating can be particularly awkward. Consider how to write a **Tutorial D** `UPDATE` statement to change student S1's mark in course C1. Consider how to write an `INSERT` statement to record that mark in the first place.
4. The playing field for queries, as with non-5NF relvars, is not level. True, some aggregation queries are slightly simplified (try obtaining the average exam mark for each course), but many others become quite complex unless a preliminary invocation of `UNGROUP` is injected (try obtaining all of student S1's exam marks).

In short, the asymmetric structure illustrated in Example 8.1 leads to asymmetric queries, asymmetric constraints, and asymmetric updates.

Now, `C_ER` is in 5NF, as you can easily verify, and it exhibits no redundancy. But see, in Figure 8.3, what happens if we apply a similar treatment to our original non-5NF relvar, `ENROLMENT`, having the attribute `Name` in place of `EXAM_MARK`'s `Mark`, giving the relvar `C_ES` with relation-valued attribute `EnrolledStudents`.



CourseId	EnrolledStudents	
	StudentId	Name
C1	S1	Anne
	S2	Boris
	S4	Devinder
	StudentId	Name
C2	S1	Anne
	StudentId	Name
C3	S3	Cindy

Figure 8.3: C_ES, derived from ENROLMENT by grouping

`C_ES` is in 5NF but exhibits exactly the same redundancy that `ENROLMENT` exhibits: in the current value of `C_ES`, student S1's name is recorded twice.

For these reasons, perhaps a group-ungroup normal form (GUNF?) could be usefully defined: relvar *r* is in GUNF if and only if no attribute of *r* is relation valued; but as far as the present author is aware no such definition is to be found in the literature (apart from this book). Codd proposed a normal form that he called **first normal form** (1NF), and he included a requirement for 1NF in his definitions for 2NF, 3NF, and subsequently BCNF. Under 1NF as he defined it, relation-valued attributes were "outlawed"; that is to say, a relvar having such an attribute was not in 1NF. However, certain examples do exist where to avoid a relation valued attribute we have to resort to artifice.

Consider, for example, the relvar, in the catalog, to record the keys of all the relvars in the database. The straightforward definition, shown in Example 8.2, involves a relation-valued attribute.

Example 8.2: Catalog relvar to record keys of relvars, not in GUNF

```
VAR RELVAR_KEYS BASE RELATION { RelvarName NAME,
                                Key RELATION { Attr NAME } }
                            KEY { ALL BUT } ;
```

We cannot obtain an equivalent design by ungrouping, because a relvar can have several keys—recall that a key can have several attributes and note that the same attribute might appear in several keys. A tuple appearing in the ungrouping, simply pairing relvar name *r* with attribute name *a* tells, us only that *a* is a member of some key of *r.* A truly equivalent design in GUNF is unachievable. The best we can do is probably as shown in Example 8.3, where we have to introduce an extra attribute. Moreover, this design does not admit relvars with empty keys (specified by `KEY { }`)—those would have to be represented by a separate relvar in the catalog.

Example 8.3: Catalog relvar to record keys of relvars, in GUNF

```
VAR RELVAR_KEYS_GUNF BASE RELATION { RelvarName NAME,
                                     KeyNumber INTEGER,
                                     KeyAttr NAME }
                                  KEY { ALL BUT } ;
```

Having to number the keys of each relvar is artificial and burdensome. Most of the noted disadvantages of relation-valued attributes are not so relevant here because we expect catalog relvars to be maintained by the DBMS. The natural, non-GUNF design of Example 8.2 is probably preferable.

Now, Codd's definition of 1NF attempted similarly to outlaw attributes of certain other types too, because relation types are not the only types that, if used for attributes of relvars, give rise to the problems identified with relation types, as we shall now see.

Wrap-unwrap normalization

Instead of defining `C_ES` with its relation-valued attribute `EnrolledStudents`, derived from `ENROLMENT` using `GROUP`, we could apply `WRAP` to derive the relvar `C_EST`, as shown in Example 8.4.

Example 8.4: Alternative design for enrolments, using a tuple-typed attribute

```
VAR C_EST BASE
    INIT ( ENROLMENT WRAP {StudentId, Name} AS SIDName )
    KEY { CourseId } ;

CONSTRAINT At_most_one_name_per_student
    COUNT ( C_EST UNWRAP SIDName { StudentId, Name } ) =
    COUNT ( C_EST UNWRAP SIDName { StudentId } ) ;

CONSTRAINT CourseId_foreign_key
    IS_EMPTY ( C_EST NOT MATCHING COURSE ) ;
```

Like `C_ER`, `C_EST` is in 5NF and yet still exhibits the same redundancy as `ENROLMENT`. Codd's definition of 1NF precluded tuple-typed attributes too. Perhaps a "wrap-unwrap" normal form (WUNF?) could be usefully defined along similar lines to the GUNF previously mooted. But even outlawing relation and tuple types wasn't sufficient for the purpose at hand. A similar effect can be obtained with user-defined types, as Example 8.5 shows.

Example 8.5: Alternative design for enrolments, using a user-defined type

```
TYPE NamedStudent POSSREP { StudentId SID, Name NAME } ;

VAR C_ESUDT BASE RELATION { CourseId CID,
                            SIDName NamedStudent }
              KEY { CourseId } ;

CONSTRAINT At_most_one_name_per_student
  WITH ( X := EXTEND C_ESUDT :
              { StudentId := THE_StudentId(SIDName) ),
                Name := THE_Name(SIDName) }
       ) :
  COUNT ( X { StudentId } ) = COUNT ( X { StudentId, Name } ) ;

CONSTRAINT CourseId_foreign_key
  IS_EMPTY ( C_ESUDT NOT MATCHING COURSE ) ;
```

Codd attempted to preclude the use of such types in his definition of 1NF, but unfortunately this definition appealed to an unclear notion of *atomicity.* To be in 1NF, a relvar's attributes all had to be of types consisting of "atomic" values only. However, he did not give a clear definition of what it means for a value to be atomic and we now believe the notion has no absolute meaning. Suffice it just to say that the relational database designer should generally avoid futile attempts, such as those shown in this section, to obtain 5NF without achieving the elimination of redundancy that 5NF is supposed to achieve. In particular, stick, where possible, to GUNF and WUNF.

8.2 Restriction-Union Normalization

If we take two restrictions of relation *r*, say *r* `WHERE` *c1* and *r* `WHERE` *c2*, and every tuple of *r* satisfies the combined condition *c1* `OR` *c2*, then we have obtained a decomposition of *r* into two relations, *r1* and *r2*, such that *r* can be reconstituted by taking the union, *r1* `UNION` *r2*. If *r* is in fact a relvar and *c1* and *c2* are such that at all times *r* must be equal to (*r* `WHERE` *c1*) `UNION` (*r* `WHERE` *c2*), then a design using *r1* and *r2* in place of *r* can be considered. However, having taken note of such possibilities, we must add right away that such decomposition, sometimes referred to as "horizontal decomposition" (appealing to our normal tabular depiction of relations), is usually contraindicated. For one thing, the restriction conditions *c1* and *c2* now have to become declared constraints on relvars *r1* and *r2*, respectively. For another, if it is possible for some tuple of *r* to satisfy both conditions, then that tuple must appear in both *r1* and *r2*, in which case the proposed decomposition introduces redundancy where there was none to start with.

For an example, consider a horizontal decomposition of `EXAM_MARK` such that we have a separate relvar for each course. Then our design would have to include the following components for course C1 and similar components for each other course, as shown in Example 8.6.

Example 8.6: Separate exam marks relvars for each course

```
VAR C1_EXAM_MARK BASE
    INIT ( EXAM_MARK WHERE CourseId = CID('C1') )
     KEY { StudentId } ;

CONSTRAINT C1_only
    AND ( C1_EXAM_MARK, CourseId = CID('C1') ) ;
```

(likewise for all the other courses)

That constraint, `C1_only`, not only ensures that every tuple in `C1_EXAM_MARK` subsequently satisfies at all times the restriction condition by which it was originally derived; it also means that `C1_EXAM_MARK` is not in BCNF! (*Exercise for the reader:* why is that so?) Now, the designer adopting such a decomposition might be tempted to dispense with the redundant attribute `CourseId` in `C1_EXAM_MARK` and all the other similarly derived course-specific relvars. In that case, `CourseId` values as such appear nowhere in the relvars for recording exam marks. How, now, can we express the query to give the course identifier and mark for every exam taken by student S1, for example? (*Exercise for the reader.*) And how can we express a constraint to the effect that each exam mark tuple must relate to one of the tuples in `IS_ENROLLED_ON`? (*Answer:* We can't.)

Although a small amount of research has been undertaken into the issues raised by horizontal decomposition, no particular restriction-union normal form has been defined and generally accepted as being useful. Suffice it, here, to say that any design involving two or more relvars with identical headings—or headings such that one can be derived from the other by invocation of `RENAME`—should be treated with suspicion and the possibility of combining those two relvars should be considered. The combination process might involve the use of `EXTEND` as well as `UNION`, as shown in Example 8.7.

Example 8.7: Reconstituting EXAM_MARK

```
VAR EXAM_MARK BASE
INIT ( EXTEND C1_EXAM_MARK : { CourseId := ( CID('C1') }
       UNION
       EXTEND C2_EXAM_MARK : { CourseId := ( CID('C2') }
       UNION
       ... ) ;
```

8.3 Surrogate Keys

Example 8.8 derives from the author's personal experience as a tutor for a certain distance learning establishment.

Example 8.8: Test paper marks

```
VAR MAX_MARK BASE RELATION { CourseId CID,
                             Year INTEGER,
                             TMA# INTEGER,
                             Question# INTEGER,
                             Max INTEGER }
       KEY { CourseId, Year, TMA#, Question# } ;

VAR STUD_MARK BASE RELATION { StudentId SID,
                              CourseId CID,
                              Year INTEGER,
                              TMA# INTEGER,
                              Question# INTEGER,
                              Mark INTEGER }
      KEY { StudentId, CourseId, Year, TMA#, Question# } ;

CONSTRAINT FK_referencing_MAX_MARK
    IS_EMPTY ( STUD_MARK NOT MATCHING MAX_MARK ) ;

CONSTRAINT Marks_no_greater_than_max
    IS_EMPTY ( ( STUD_MARK JOIN MAX_MARK )
               WHERE Mark > Max ) ;
```

The abbreviation TMA stands for tutor-marked assignment. Each year, for each course being presented, a certain number of TMAs are devised as test papers for students. A tutor is assigned to each student and one of the tutors' duties is to mark their students' TMA submissions. I imagine Example 8.8 as a possible design underpinning the process I, as a tutor, have to follow when I do the marking. `MAX_MARK` gives the score out of which each question must be marked. `STUD_MARK` gives each student's mark for each question.

Now, some designers would worry about the repetition of attributes `CourseId`, `Year`, `TMA#`, and `Question#` in the two relvars of Example 8.8, these attributes constituting the foreign key defined by the constraint `FK_referencing_MAX_MARK`. (In the actual scenario from which this example was derived the situation is exacerbated because questions are divided into separately marked parts and subparts, adding two more attributes to both keys.) Somehow it would seem more efficient if `MAX_MARK` had a key consisting of just one attribute, and then `STUD_MARK`'s foreign key would in turn consist of a single attribute corresponding to that singleton key in `MAX_MARK`.

Example 8.9: Test paper marks, using a surrogate key

```
VAR MAX_MARK BASE RELATION { QuestionId QID,
                             CourseId CID,
                             Year INTEGER,
                             TMA# INTEGER,
                             Question# INTEGER,
                             Max INTEGER }
        KEY { CourseId, Year, TMA#, Question# }
        KEY { QuestionId } ;

VAR STUD_MARK BASE RELATION { QuestionId QID,
                              StudentId SID,
                              Mark INTEGER }
               KEY { QuestionId, StudentId } ;

CONSTRAINT FK_referencing_MAX_MARK
    IS_EMPTY ( STUD_MARK NOT MATCHING MAX_MARK ) ;

CONSTRAINT Marks_no_greater_than_max
    IS_EMPTY ( ( STUD_MARK JOIN MAX_MARK )
                WHERE Mark > Max ) ;
```

RUSSIAN ENTREPRENEUR
LUXURY GOODS MARKET LONDON
INDIAN NUCLEAR POWER CORP
US FIRM OUTSOURCING
AFRICA HEALTH
CHINA PAYS PREMIUM
TROUBLED LONDON B
WHILE YOU WERE SLEEPING...
DUKE THE FUQUA SCHOOL OF BUSINESS
www.fuqua.duke.edu/whileyouweresleeping

The singleton key of MAX_MARK is called, slightly inappropriately, a **surrogate key**. (A surrogate is a person or thing taking the place of another. The key in question here must clearly sit alongside, not in place of, the one that must be specified in any case—a point sometimes overlooked by advocates of surrogate keys. The thing that really "take[s] the place of another" is the *foreign* key attribute of STUD_MARK.)

Now, it might well be the case that the performance characteristics of the DBMS are such that the design in Example 8.9 speeds up most of the anticipated tasks in maintaining and using the database. But in general the technique should be viewed circumspectly, taking into account the following considerations, possibly negating those perceived performance gains:

- Example 8.9 involves an extra key constraint.
- Many queries on STUD_MARK need a join with MAX_MARK in Example 8.9, not needed in Example 8.8.
- The values for the compound key of MAX_MARK arise in the real world. Those for the surrogate key have to be "magicked" from somewhere. That process probably has to be automated, thus incurring overhead not present in Example 8.8. (Some commercial DBMSs do provide facilities for such automation.)
- When a tuple is added to STUD_MARK in Example 8.9, how is the required value for QuestionId to be determined? Don't we have to search MAX_MARK for the CourseId, Year, TMA#, and Question# values, all of which must be to hand, to obtain the corresponding QuestionId value?

The best advice concerning use of surrogate keys might well be: if in doubt, don't use them (and if not in doubt, reconsider!).

8.4 Representing "Entity Subtypes"

Practitioners of Entity-Relationship modelling, sometimes used as a preliminary stage in database design, make a distinction between real world entities and the relationships among those entities. Entities are classified into "types", but an entity type is not a type in the sense used elsewhere in this book. For example, the tuples of IS_CALLED and COURSE might be considered to represent "student entities" (entities of type "student") and "course entities" (entities of type "course"), respectively, whereas those of IS_ENROLLED_ON represent relationships between certain student entities and certain course entities. An entity subtype arises when some entities of a given type *ET*, but not all, are perceived to have something in common that merits the subset consisting of those particular entities being identified as an entity type in its own right, this being a *subtype* of *ET*.

For example, consider payments debited from a bank account. Each can be identified by an account number and a transaction number and includes an amount of money to be debited from the account in question and some identification of the payee. Some payments are made by cheque; these, and only these, have a cheque number in addition to the information common to all payments. Others are made under direct debit arrangements; these, and only these, have additional information concerning the arrangement under which they are made. Yet others are made using a debit card; these, and only these, include a debit card number (for there can be more than one debit card for a joint account). And so on.

The designer of the database where payments against bank accounts are to be recorded is faced with the following choice:

Option 1: Have separate relvars for each payment type, each with attributes for what are common to all payments.

Option 2: Have just one relvar for the common attributes and separate relvars for the different payment types, each with attributes sufficient to identify the transaction, plus attributes for the data unique to its own payment type.

If Option 1 is adopted, then a "disjointness" constraint is needed to ensure that no account number/transaction number combination appears in more than one participating relvar. In the example at hand, the situation is further complicated by the fact that for a debit card payment the demands of BCNF mean that the relevant account number must be stored in a separate relvar, as mentioned in Chapter 7, Section 7.8, under the heading **FD Loss Sometimes Inevitable**. (Recall that a payment by debit card is identified by a transaction number and a debit card number, the account number being implied by the debit card number.)

If Option 2 is adopted, then we need a foreign key constraint on each "subtype" relvar to ensure that every payment of the type in question is indeed in connection with a transaction recorded in the "parent" relvar. We will also need a disjointness constraint on the subtype relvars, and a constraint to ensure that for each tuple in the parent relvar there is a matching tuple in one of the subtype relvars.

It seems, then, that on the evidence so far Option 1 might be preferable for being less complicated. And that brings me back, for the last time, to `WIFE_OF_HENRY_VIII`. The suggested decomposition into `W_FN_LN` with attributes `Wife#`, `FirstName`, and `LastName`, and `W_F` with attributes `Wife#` and `Fate` can be seen as an implementation of Option 2. Option 1 could be used instead, where `W_F` is replaced by a relvar with the same heading as the original `WIFE_OF_HENRY_VII`. Now, if Option 1 is chosen, then only the current wife's marriage is missing a `Fate` value. In that case no more than one tuple can ever appear in `W_FN_LN` and the empty set, not {`Wife#`}, is the only key of that relvar. Thus both of the examples discussed in this section—bank account debits and King Henry's wives—show that when several relvars can be perceived as representing subtypes of some common entity type, they do not necessarily have a key in common, contrary to what is stated in some texts on this subject. In the first example, payments by debit card show that they don't even have to have a superkey in common.

The choice between Option 1 and Option 2 might be influenced by updating issues. The situation is somewhat similar to that described in connection with Example 7.7 in Chapter 7. With Option 1 a multiple assignment is needed to update both relvars simultaneously when the king's current marriage is terminated; with Option 2 one merely adds a tuple to `W_F`. That particular issue doesn't arise with the bank account example if we can assume that a payment never changes its type.

Appendix A: References and Bibliography

In addition to the books and papers referenced in the body of the book I include three textbooks ([3], [7], and [14]) that I have chosen from my own collection.

[1] http://en.wikipedia.org/wiki/Database

The article that caused me to reference Wikipedia in Chapter 1 has been significantly revised and no longer includes the text I cited. The article should be treated with caution. For example, in case it still refers to "Date's Information Principle", please note that the principle in question actually originates from E.F. Codd.

[2] W.W. Armstrong: "Dependency Structures of Data Base Relationships", Proc. IFIP Congress, Stockholm, Sweden (1974).

[3] Paolo Atzeni and Valeria De Antonellis: *Relational Database Theory.* Redwood City, Ca.: The Benjamin/Cummings Publishing Company, Inc. (1993).

The Preface begins, "This book presents a systematic treatment of the formal theory of the relational model of data." Here you will find many of the topics of the present book—and some more—treated formally. You will need a good grounding in logic. Plenty of exercises. 386 pages.

[4] Luca Cardelli and Peter Wegner: "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Comp. Surv. 17,* No. 4 (December 1985).

[5] E.F. Codd. "A Relational Model of Data for Large Shared Data Banks", *CACM 13,* no. 6 (June 1970). Republished in "Milestones of Research", *CACM 25,* No. 1 (January 1982).

Usually cited as the seminal paper that first introduced the relation model of data. Actually, Codd originally published his idea a year earlier, in an IBM research report.

[6] E.F. Codd: "Recent Investigations into Relational Data Base Systems", Proc. IFIP Congress, Stockholm, Sweden (1974) and elsewhere.

[7] Thomas M. Connelly and Carolyn E. Begg: *Database Systems, A Practical Approach to Design, Implementation, and Management* (3rd edition). Harlow, England: Pearson Education Limited (2002).

A massive tome and a popular recommendation for the serious student or practictioner. 1236 pages.

[8] Hugh Darwen, C.J. Date, and Ronald Fagin. "A Normal Form for Preventing Redundant Tuples in Relational Databases", Proc. 15th International Conference on Database Theory, Berlin, Germany (March 26th-29th, 2012).

A PDF copy is available at www.edbt.org/Proceedings/2012-Berlin/papers/icdt/a02-Darwen.pdf. The paper describes a normal form—Essential Tuple Normal Form (ETNF)—that is weaker than 5NF but nevertheless sufficient, and in fact necessary, for avoiding the kind of redundancy that can arise from join dependencies. As a matter of fact this paper arose during the drafting of Chapter 7, when I came across the problem mentioned in the annotation to reference [10] below.

[9] C.J. Date: *An Introduction to Database Systems* (8th edition). Reading, Mass: Addison-Wesley (2004).

Perhaps the best known book on the subject, by the best known author. A *sine qua non* for all database professionals in industry or academe. 983 pages. (But beware of its definition of 5NF, which is not quite correct. For one thing, it fails to take account of JDs containing redundant projections, but more significantly, as noted in reference [8], it actually defines a slightly weaker NF than 5NF, though still stronger than reference [8]'s ETNF and thus still sufficient but not necessary.)

[10] C.J. Date: *The Relational Database Dictionary* (extended edition). Berkeley, Ca.: Apress (2008).

An invaluable booklet for quick reference. (But again, beware of its definition of 5NF. Its condition (a) propagates the error of [9] and its condition (b) is also erroneous.)

[11] C.J. Date and Hugh Darwen: *Database Explorations: Essays on* The Third Manifesto *and Related Topics*. Bloomington, IN: Trafford Publishing (2010). See http://www.trafford.com/Bookstore.

[12] C.J. Date and Hugh Darwen: *Databases, Types, and The Relational Model:* The Third Manifesto (3rd edition). Reading, Mass: Addison-Wesley (2007).

Related material, including a complete grammar for **Tutorial D**, can be found at www.thethirdmanifesto.com.

[13] Ronald Fagin: "Normal forms and relational database operators", Proc. 1979 ACM-SIGMOD (ed. P.A. Bernstein) 153–160. Available at

http://www.almaden.ibm.com/cs/people/fagin/sigmod79.pdf

Here, slightly paraphrased, is the algorithm referred to in Chapter 7, Section 7.4, for determining whether a given JD is implied by the keys of the applicable relvar.

Given a relvar *R*, a set *S* of keys {*K1, …, Kn*} of *R,* and a JD *{*P1, …, Pm*}: Initialize set *T* as {*P1, …, Pm*}. Apply the following rule until it can no longer be applied: If *Ki* is a subset of the intersection of *Y* and *Z* for some *i* (1⩽*i*⩽*n*) and for distinct member *Y* and *Z* of *T,* then replace *Y* and *Z* in *T* by the set that is the union of *Y* and *Z* (so the number of members of *T* decreases by one). If, when the algorithm terminates, some member of *T* contains every attribute of *R*, then the algorithm *succeeds* and the given JD is implied by the keys of *R*; otherwise the algorithm *fails* and the given JD is not implied by the keys of *R*.

[14] Lex de Haan and Toon Koppelaars: *Applied Mathematics for Database Professionals.* Berkeley, Ca.: Apress (2007).

An "interestingly different" book on the theory and practice of relational databases that I personally recommend. The book includes an excellent treatise on the design of procedural methods for overcoming the deficiencies, in the area of integrity checking, of current SQL implementations. These deficiencies are mentioned in Chapters 6 and 7of the present book.

[15] I.J. Heath: "Unacceptable File Operations in a Relational Database", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Calif. (November 1971).